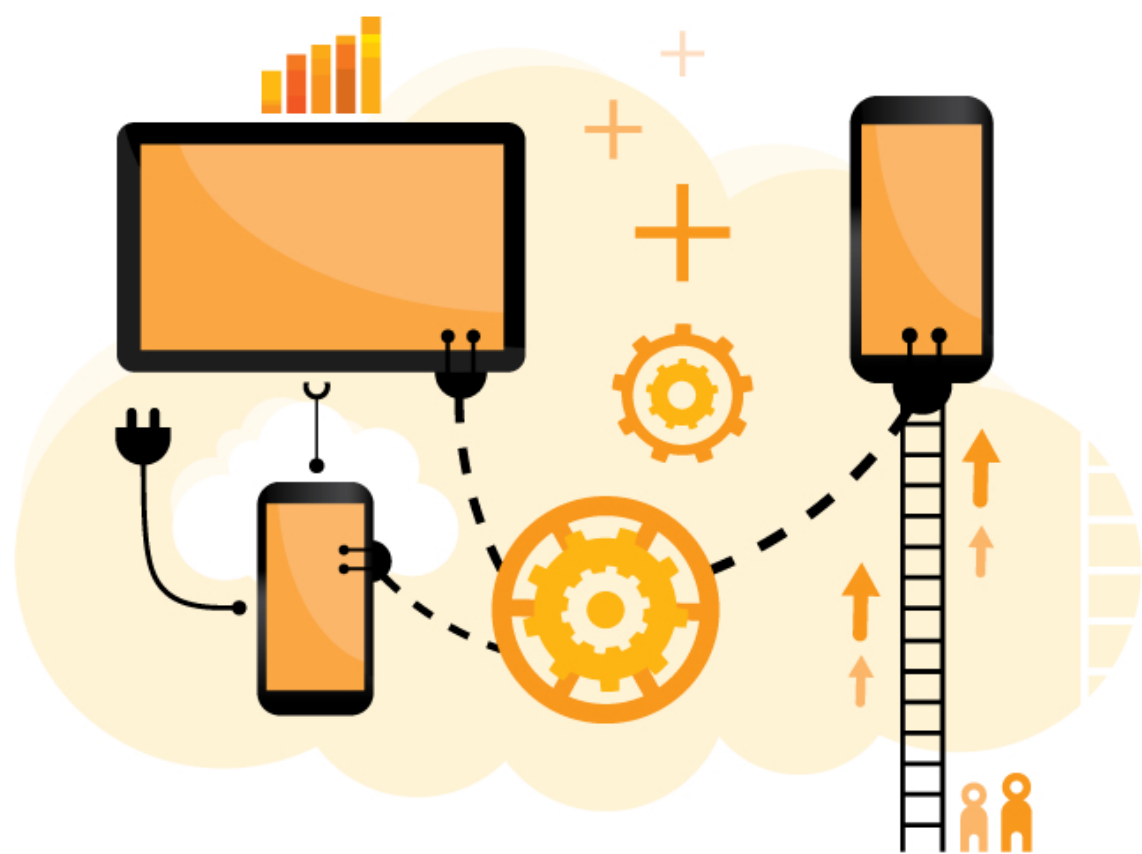
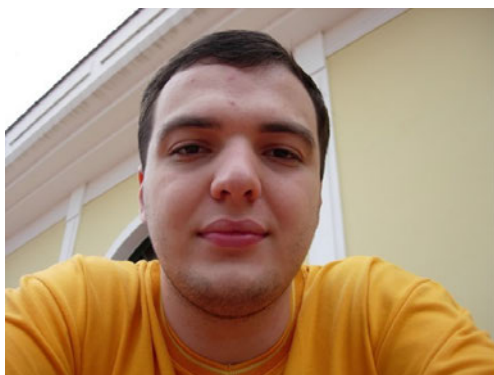


A Web Mobile

Programa para um mundo de muitos dispositivos



Sobre mim



Escrevi minha primeira linha de código com 14 anos em 1999 e ela foi em HTML. Daí pra CSS e JavaScript foi um pulo. Em seguida, me aventurei em SSI e PHP, incluindo bancos de dados. Em 2003, iniciei meu curso de Ciência da Computação na USP e nadei em águas mais profundas desde então — Java, C, Python. Cresci bastante em programação backend.

Mas eu sempre fui apaixonado por front-end.

Com o renascimento da Web através do HTML5 nos últimos anos, voltei a focar na minha paixão. Respiro front-end o dia todo. Leio muito, estudo muito, escrevo muito e programo muito — desde que envolva bastante HTML, CSS e JavaScript.

E, de algum tempo pra cá, resolvi focar em *mobile*. Meu primeiro site mobile eu escrevi há quase uma década usando WML pra redes WAP (se você é novo, talvez precise da Wikipedia pra entender a frase anterior). Imagine minha animação então nessa nova era dos smartphones e o que significam para a Web. Acredito muito na Web única como plataforma democrática e universal.

Já trabalhei em algumas empresas, programando em várias linguagens (já até

ganhei dinheiro com opensource). Desde 2004, trabalho na **Caelum** como instrutor e desenvolvedor. Foi onde minha carreira decolou e onde mais aprendi, e aprendo todo dia. É onde pretendo passar ainda muitos e muitos anos.

Ensinar e escrever são uma paixão desde o colégio — lembro a decepção da minha professora de português quando ela descobriu que eu seguiria carreira em exatas. Dar aulas, escrever artigos, blogar e palestrar são minha maneira de misturar essas habilidades.

Esse livro é ponto alto em toda essa trajetória. Espero que seja divertido pra você ler tanto quanto foi, para mim, escrever. Obrigado por acreditar nele e comprá-lo.

Você pode me encontrar também escrevendo por aí na Web:

- Meu blog pessoal onde escrevo bastante sobre Web, mobile, front-end em geral: <http://sergiolopes.org>
- O blog da Caelum, onde sempre publico artigos sobre front-end: <http://blog.caelum.com.br>
- Meu Twitter e meu Facebook onde posto muitos links pra coisas bacanas de front-end e mobile: <https://twitter.com/sergio-caelum> e <https://www.facebook.com/sergio.caelum>
- E também participo de vários fóruns, grupos e listas de discussão de Web, onde a gente pode se encontrar. Meu favorito é o novo GUJ: <http://www.guj.com.br/perguntas>

E, se nos toparmos um dia em algum evento, não deixe de me chamar pra batermos um papo.

— Sérgio Lopes, 2013

Sumário

1	A Web Mobile	1
	Estratégia mobile	5
2	Os caminhos de uma estratégia mobile	7
3	App ou Web? Comparativo de possibilidades	13
4	HTML 5 é diferente de HTML 5: o caso das packaged apps	21
5	Design Responsivo por uma Web única	27
6	Mobile-first	33
7	Mercado, Browsers, suporte e testes	39
	Programando a Web moderna	47
8	Flexibilidade na Web com layouts fluídos	49
9	Media queries: as melhores companheiras de um layout fluído	59
10	Tudo que você queria saber sobre viewport	67
11	A saga dos 3 pixels e as telas de alta resolução e retina	79
12	Não remova o zoom dos seus usuários	87
		iii

13 Use sempre media queries baseadas no conteúdo da sua página	91
14 Media queries mobile first ou desktop first?	95
15 As media queries para resoluções diferentes e retina	101
16 Media queries também ajudam na acessibilidade	105
 A Web adaptativa	 111
17 Progressive enhancement e feature detection	113
18 RESS — Design responsivos com componentes no lado do servidor	119
19 O complicado cenário das imagens responsivas	129
20 Design adaptativo e carregamento condicional	143
 Gestos e entrada de dados	 153
21 Gestos na Web	155
22 Implementando gestos com JavaScript	161
23 Desafios de UX em interfaces touch	173
24 Use os novos input types semânticos do HTML 5	183
25 Revisitando os antigos componentes de formulários	199
26 Usabilidade de formulários mobile	207
 Conclusão	 215

CAPÍTULO 1

A Web Mobile

Eu não gosto do título desse livro. “*A Web Mobile*”. Quer dizer, é um título bacana e fui eu mesmo que dei. Mas não fiquei satisfeito. Ele só conta uma parte da história do livro.

Fato é que não existe uma *Web mobile*. Existe *a Web*, que a gente acessa tanto do computador quanto do celular ou do tablet. É o mesmo HTML, CSS e JavaScript que rodam nos navegadores independente do dispositivo. Eles têm diferenças entre si, claro, mas a maior parte é a mesma coisa. É tudo *Web*.

Vários tópicos do livro não são específicos de dispositivos móveis. *Media queries*, por exemplo, um assunto que você talvez já tenha visto sempre ligado a sites mobile; aqui no livro, quando chegarmos nelas, vamos ver até como afetam a acessibilidade no Desktop. Ou ainda quando falarmos de *touch screens*: não dá pra chamar isso de *mobile*, já que computadores com touch são cada vez mais comuns hoje.

Pensei em dar o título pro livro de “*A Web moderna*”, ou algo assim. Mas seria muito genérico, afinal o tema central aqui acaba sendo *mobile* mesmo.

O livro é sobre como programar a Web atual, com tecnologias novas, pensando

nos dispositivos de hoje e de amanhã. E muitas das transformações recentes da Web foram causadas pelas revoluções e novidades dos dispositivos móveis. Então, não é que o título do livro está incorreto, mas, talvez, incompleto. Deveria ser algo como *“A Web moderna e como ela foi afetada pelos dispositivos móveis”*. Acabei resumindo para *“A Web Mobile”*.

O mercado mobile

Decidi que não ia começar o livro citando estatísticas de como o mercado mobile cresceu muito nos últimos anos e como ele vai estourar nos próximos. Nem mostrar a evolução do 3G ou 4G no Brasil ou falar de quanto de participação no mercado cada plataforma móvel tem.

Qualquer número desses ficaria desatualizado pouco depois da publicação. E também acho que, se você está lendo esse livro, é porque não precisa ser convencido de que **o mercado mobile tem um potencial gigantesco**.

Se você precisar de informações detalhadas sobre o crescimento desse mercado, comportamento dos consumidores e outros dados, recomendo o **Our Mobile Planet** do Google (<http://www.thinkwithgoogle.com/mobileplanet/pt-br/>). Ele tem dados do mundo todo, inclusive de pesquisas específicas do Brasil.

Outra fonte onde você consegue números atualizados mês a mês é o **Stat Counter**, sobre acesso a Web (<http://gs.statcounter.com/>). E os sites de notícias de tecnologia estão sempre mostrando matérias atualizadas com novos dados e pesquisas de mercado — Info, Techtudo, Gizmodo Brasil.

Fato é que o mercado para dispositivos móveis é gigantesco e cheio de oportunidades. Muita gente está se dando bem: fabricantes de aparelhos, operadoras de telefonia, criadores das plataformas, desenvolvedores de aplicativos e os que acreditam no acesso a Web via mobile. O foco do livro é nesse último grupo. **É para todos que querem explorar o potencial de se ter um navegador de Internet no bolso de milhões de pessoas.**

O livro

Organizei o livro como uma série de tópicos com temas diversos. Não é um livro para iniciantes; você precisa ter familiaridade com Web e HTML, CSS e JavaScript. Também não é um livro com o qual você vai desenvolver um projeto prático do início ao fim — recomendo o livro *“Web Design Responsivo”* do amigo Tércio Zemel se você quer um passo a passo mais prático com projetinho (<http://www.casadocodigo.com>).

br/products/livro-web-design-responsivo).

Esse livro é como uma coleção de artigos sobre Web, mobile e assuntos relacionados. Cada seção em geral é curta e pode ser lida independentemente das outras. Você pode até ler fora de ordem ou pular alguma que não lhe interesse.

Em diversos momentos, vou mostrar exemplos práticos pra você testar. Além do código mostrado, você também pode executar direto no seu dispositivo acessando as URLs de exemplo ou os *qr codes* que espalhei pelo livro. Se estiver lendo o ebook, todas as referências são clicáveis; para os leitores do impresso, coloquei os links de todas as referências em <http://sergiolopes.org/livro-web-mobile/referencias.html> pra facilitar.

Apesar de ser um livro bem técnico e prático, eu tomei a liberdade de dar minhas opiniões pessoais em alguns momentos. Tento deixar claro sempre que algo é uma opinião pra você analisar, julgar e discordar de mim à vontade.

Existe um grupo de discussões oficial do livro no Google Groups pra você tirar dúvidas e discutir comigo e outros leitores (<https://groups.google.com/d/forum/livro-web-mobile>). E tem o GUJ Perguntas (<http://www.guj.com.br/perguntas/>) onde você também pode postar seus questionamentos.

Esse não é um livro completo. No *brainstorm* original que fiz, acabei com uma lista de assuntos para uns 10 volumes diferentes. Tive que escolher o que ia entrar aqui e o que ia ficar de fora. Gostei da seleção final de tópicos mas talvez eu tenha deixado de fora justo *aquele* assunto que você queria saber. Quem sabe um dia eu escreva os outros 9 volumes mas, até lá, você pode me encontrar no meu blog onde escrevo mais: <http://sergiolopes.org>

Boa leitura!

Parte I

Estratégia mobile



CAPÍTULO 2

Os caminhos de uma estratégia mobile

Então você, convencido do potencial do mercado mobile, decide atacá-lo. Por onde começar? O primeiro ponto é discutir os caminhos possíveis e, então, traçar uma *estratégia mobile*.

Porque mobile?

O primeiro ponto é definir o motivo de nossa empresa ou projeto encarar o mundo mobile. Que objetivos queremos atingir? Qual o público-alvo e do que ele precisa?

Toda reunião que começa com a frase “*precisamos de uma App pro iPhone*” já começou errada. A App ou o site mobile são meios pra um fim maior. Que objetivos meu usuário quer atingir com essa App?

Esse levantamento do porquê é importante para definir seus passos no mundo mobile. Dependendo do resultado, podemos concluir que uma App nativa para

iPhone é o melhor. Ou que o melhor é um site mobile. Ou que não precisamos de nada disso.

O site da Apple, a mãe da era mobile moderna, até hoje não tem uma versão mobile. E não tem App pra iPhone também, nem pra iPad. Nos levantamentos deles, provavelmente concluíram que o site Desktop bem construído é suficiente pra todo mundo. Um tanto irônico, mas um bom exemplo de que montar uma App ou site mobile só pelo oba-oba não é uma boa ideia.

App ou Web?

Uma vez decidida a investida em uma presença mobile, a primeira resolução necessária costuma ser se criamos uma App ou se investimos na Web mobile. O próximo tópico do livro (3) vai se aprofundar nas diferenças práticas e técnicas dessas duas abordagens. Depois, no tópico 4, vamos discutir Apps híbridas construídas com tecnologias da Web.

As diferenças comumente citadas de performance, acesso a recursos de hardware e etc eu nem considero tão relevantes assim. Agora, quero focar na discussão mais *estratégica* por traz dessa decisão. Os aspectos mais técnicos, aliás, muitas vezes, são irrelevantes. Apesar de existirem diferenças, sim, de performance entre nativo e Web, elas não são determinantes para a maioria das aplicações e usuários. A Web é *boa o suficiente* para a maior parte dos cenários, assim como ela é boa o suficiente no Desktop apesar de perder em performance para softwares nativos por lá também.

Expectativas

Na minha opinião, o aspecto que mais deve pesar na decisão por App ou Web é a expectativa do usuário com relação a sua empresa, seu projeto, sua marca.

Se você está criando um novo produto ou nova empresa, pensando em algo inovador, pode não fazer muita diferença se é uma App ou Web. Você ainda não tem usuários, então não há expectativas com relação a seu produto. Você pode criar algo novo e forte em uma direção específica e não perder ninguém.

Um exemplo prático: o *Instagram* nasceu como uma App para iPhone e só. Era um novo serviço com objetivo de explorar justo o nicho de compartilhamento de fotos no smartphone da Apple. Estrategicamente, a empresa focou num certo nicho e inovou com um produto diferente. Não tinha uma App para Android, um site mobile e nem mesmo uma versão da App pra iPad. Era um foco estratégico e de negócios que só uma empresa nova podia ter. Depois, com o crescimento, lançaram

a versão para Android mas nada muito além disso.

Agora, se você já tem uma presença na Web, um produto consolidado, iniciar sua estratégia mobile com uma App nativa pode ser um tiro no pé. Imagine um portal de notícias nacional decidir entrar no mundo mobile lançando uma App pra iPhone. Além de deixar a maior parte do mercado de fora, é uma situação que pode ser anormal até para o próprio usuário do iPhone. O portal é tão forte na Web que o usuário está acostumado a ler as notícias no navegador e, provavelmente, deve abrir o site mesmo no iPhone.

Qual é a expectativa do usuário?

Web first

Eu gosto muito da Web. Acredito demais nela como plataforma portátil e democrática. Escrevi esse livro sobre Web! Mas, apesar de tudo isso, concordo que Apps têm seu lugar no mundo, claro. Há muitos cenários onde uma App traz melhor experiência para o usuário e satisfaz melhor suas expectativas, sobretudo com relação à usabilidade.

Mas o mercado de Apps hoje é exclusivo, e apostar em uma plataforma X é arriscado. Alguns anos atrás, apostar no iOS parecia o tiro certo para atingir a maioria do mercado. Hoje, o Android é dominante na maior parte do mundo (e ganha de lavada no Brasil). E amanhã?

Muita gente que apostou numa App iOS há alguns anos está agora correndo atrás da versão Android. E vão precisar correr atrás de qual plataforma depois? Já quem apostou na Web está tranquilo.

Uma estratégia é começar sempre pela versão Web do seu produto ou webapp. Sedimentar bem sua presença mobile via Web, garantindo acesso universal e multiplataforma. Aí, conforme as necessidades surgirem e seu planejamento financeiro permitir, você pode investir em Apps específicas de plataformas com recursos e experiências nativas. *Web first*.

Várias grandes empresas da Web seguiram esse caminho. Facebook, Google e Twitter são exemplos óbvios. O Facebook tem Apps para iOS e Android bem integradas às plataformas, mas elas só foram construídas depois do site mobile que até hoje oferece suporte a todo tipo de plataforma móvel. Aqui no Brasil, portais de notícias grandes seguiram a mesma estratégia — UOL, Globo, etc —, assim como a maioria das lojas virtuais.

Em comum a todos eles? Já eram uma marca forte na Web e o caminho mais lógico era suportar mobile também com a Web. É o que os usuários deles esperam.

E, depois, se necessário, Apps específicas foram criadas.

A Web Mobile

Aqui no livro vamos claramente seguir o caminho da Web. Mas, logo de cara, já temos uma outra decisão importante a tomar: vamos de único site responsivo ou de site mobile específico?

Do ponto de vista do usuário, é uma decisão não muito importante. Não interessa tanto para ele se está acessando um site totalmente reescrito para mobile ou se é a mesma versão, desde que ele atinja seus objetivos. A decisão é muito mais técnica, e vamos discutir bastante sobre ela nos tópicos 5 e 6.

Uma coisa importante para ter em mente são as diferenças na usabilidade entre dispositivos móveis e Desktops. O papa da usabilidade, Jakob Nielsen, fala que as diferenças são tão brutais que precisamos de *designs diferentes* para atacar esses públicos. Isso pode ser feito de várias maneiras: sites diferentes pra mobile e Desktop; servidor otimizando a página (RESS, tópico 18); ou design responsivo e adaptação do design no cliente.

De qualquer maneira, ele fala (<http://m.netmagazine.com/interviews/nielsen-responds-mobile-criticism>):

“Desde que cada usuário veja o design apropriado, a escolha entre essas opções de implementação deve ser uma decisão de engenharia e não uma decisão de usabilidade.”

Vamos falar bastante no livro sobre as diferenças de usabilidade entre mobile e Desktop, em especial na parte 4. A decisão técnica preferida será pelo design responsivo, apesar de apresentar cenários mistos de adaptação com RESS (tópico 18) e carregamento condicional (tópico 20).

O principal argumento a favor do design responsivo é a simplificação do desenvolvimento. Um único projeto, um só código, um só conteúdo — mas, claro, com as devidas adaptações de design.

Design responsivo é também a **recomendação oficial do Google** para os sites a serem indexados por eles (<https://developers.google.com/webmasters/smartphone-sites/details>). Uma única URL servindo a todos os usuários é o melhor para SEO e para a experiência dos usuários.

Conclusão

A Web como plataforma única e portátil é a solução mais adequada para uma estratégia mobile democrática e acessível. Sempre que possível, comece sua inves-

tida no mobile pela Web (*Web first*) usando *design responsivo*. Cuide das adaptações necessárias no design e usabilidade dependendo do contexto de uso, e foque na experiência e nas expectativas do usuário.

CAPÍTULO 3

App ou Web? Comparativo de possibilidades

Eu sei que esse livro é sobre *Web*, então há uma clara tendência minha e do grupo de leitores para esse caminho. Mas nem sempre essa escolha é tão simples, e acho que vale uma breve discussão técnica. E, mesmo se você já tiver se decidido por um caminho, é bom analisar os prós e contras dessa decisão.

A grande diferença entre Apps e a Web que geralmente se discute é que uma App dá melhor acesso e integração ao hardware e à plataforma nativa do aparelho, enquanto que a Web traz independência de plataforma e portabilidade. Mas existem milhões de detalhes aí no meio que precisam ser discutidos.

Integração com hardware e plataforma

Uma App tem acesso direto ao hardware do aparelho e a recursos do sistema operacional. Consegue se integrar com funções avançadas e a outras Apps. Pode manipular o funcionamento do aparelho e até substituir ou complementar funções

nativas.

Já a **Web roda enjaulada dentro do navegador** e, por razões de segurança, não tem acesso direto à plataforma nativa. Mas existem diversas APIs novas do HTML 5 que expõem acesso a recursos antes exclusivos das Apps — exemplos clássicos são acesso à câmera, geolocalização, informações de acelerômetro e giroscópio, e animações 3D com aceleração na GPU. Mas, claro, há muitas coisas mais específicas que ainda só as Apps têm acesso e que não são possíveis pela Web.

O que é preciso definir aqui é que tipo de requisito você tem. Se você precisar de coisas muito específicas, uma App será o caminho. Mas as capacidades da Web são suficientes para grande parte dos cenários.

Segurança e privacidade

Rodar dentro do navegador tem suas vantagens. As restrições de segurança são fortes e a chance de acontecer algo de ruim é bem pequena. O usuário está mais protegido abrindo um site Web do que instalando uma App em seu aparelho.

As lojas de Apps tentam minimizar o impacto ruim na segurança com restrições e permissões explícitas que o usuário tem que aprovar. Mas a verdade é que a maioria dos usuários não compreende os impactos das permissões que aprova, ainda mais quando a lista é grande e cheia de termos estranhos (como na instalação de uma App no Android).

Outra maneira de tentar evitar problemas são os mecanismos de aprovação das lojas como da Apple que fazem um pente fino. Ou ainda o sistema de detecção de código malicioso que o Android tem. Mas existem vários casos de tudo isso sendo burlado, e Apps que roubam dados pessoais vão parar na loja do iOS e vários vírus na do Android.

Isso tudo pelo ponto de vista do usuário, que prefere estar mais protegido. Do ponto de vista do desenvolvedor, o cenário pode ser o inverso: se você quer uma App que acesse os dados do usuário e tenha altos privilégios, a Web vai limitá-lo. Mas, pro usuário, Web é mais segura.

A Web não é isenta de problemas, claro. Mas as décadas de evolução dos navegadores fez tudo ser mais seguro. Mais limitado também, mas por boas razões de segurança e de privacidade.

Performance

Apps sempre serão mais rápidas que a Web. Elas rodam direto no sistema opera-

cional e, na maioria dos casos, são escritas nativamente para a plataforma específica, o que dá muita performance. A Web roda dentro do navegador, que interpreta seu HTML, CSS e JavaScript, um processo relativamente mais lento.

Mas poucas aplicações *realmente precisam* de performance absurda. Não me entenda mal: não estou dizendo que é tudo bem ser lento. De modo algum. O ponto é que, para muitos casos, a diferença de performance entre a Web e uma App nativa não é perceptível para o usuário. Os navegadores de hoje são excelentes e cada vez mais rápidos, e só perdem de forma perceptível em casos bem específicos que exigem alta performance.

A maior diferença de performance para o usuário não é a execução do código em si, mas o carregamento inicial. A página Web precisa ser baixada do servidor com todas as suas dependências, o que pode demorar. Mas, claro, uma App precisa ser instalada, o que pode ser um processo mais lento ainda.

Usabilidade e visual

Um aspecto bastante citado como vantagem das Apps é a integração visual da mesma com a plataforma em si. Toda a experiência de uso da plataforma, sua usabilidade e componentes visuais, são transportados pra App. Ela tem **a cara da plataforma e é familiar para o usuário**. Pense naquele estilo característico dos botões do iOS, por exemplo, que seriam um frankenstein se usados no meio de uma App Android.



Figura 3.1: Exemplos de botões e navegação do iOS.

E não é só visualmente; também há diferenças de usabilidade. Uma frequentemente levantada é com relação ao **botão de voltar**. O iOS não tem botão de voltar e cada tela de cada App precisa prover seu próprio *voltar*. O Android e o Windows Phone já têm o botão voltar nativo, seja fisicamente no aparelho ou virtualmente na barra de baixo do sistema. É estranho fazer um botão de voltar numa App Android, assim como não se deve fazer uma App iOS sem botão de voltar.

Não há certo ou errado. O que há são diferenças de usabilidade das plataformas e diferenças na expectativa do usuário de cada uma. Apps vão dar a possibilidade de criar uma experiência nativa completa e específica para cada ambiente.

E a Web? Algumas coisas já se resolvem sozinhas — o problema do botão de voltar não existe já que todo navegador (iOS ou Android) já inclui ele lá. Para as outras diferenças visuais, você até pode criar um CSS pra cada plataforma, mas isso é meio inviável e pode gerar resultados bizarros (tentar recriar componentes nativos em CSS é um prato cheio pra cair num caso de *Vale da Estranheza* — http://pt.wikipedia.org/wiki/Vale_da_estranheza).

O mais comum é ter uma **linguagem visual única na Web**, não atrelada a nenhuma plataforma específica. É como a Web sempre funcionou no Desktop. Os sites ou webapps costumam ter **um estilo mais ligado à identidade visual da marca e da empresa**, do que da plataforma de acesso.

Pegue o exemplo dos emails. Se você abrir o Outlook no Windows vai ver um programa nativo com cara de Windows. Se abrir o Apple Mail no Mac OS X, cara de Mac. Mas, em ambos os casos, você pode abrir o Gmail no navegador e não vai se sentir nem no Windows nem no Mac; vai se sentir no Gmail, com cara de Gmail, uma identidade visual única.

Eu não vejo problema em transpor esse mesmo conceito pra Web móvel. Muita gente discorda e acha que as Apps mobile precisam estar atreladas ao visual de cada plataforma. Eu prefiro pensar que, nos últimos 20 anos de Web, os usuários aprenderam a não esperar nenhum tipo de uniformidade visual nas páginas que acessam, e até passaram a gostar dessa diversidade.

Mas eu sei que o tema é polêmico. Se você é do time que prefere o visual nativo, não feche o livro agora, só considere essa ideia de que a expectativa de um usuário da Web é mais relaxada nesse quesito.

Instalação e distribuição

A diferença é bem óbvia aqui, mas vale discutir as implicações. Uma App precisa ser instalada e isso, geralmente, envolve uma loja do fabricante onde você vai disponibilizar sua App. Existe todo o processo lento e burocrático de ser um desenvolvedor cadastrado na plataforma (e pagar por isso), além de ter que submeter suas Apps pra aprovação em muitos casos — iOS sendo notoriamente o caso mais implicante.

Para o usuário, ele precisa entrar na loja em seu dispositivo, buscar sua App, clicar em instalar, esperar a instalação e depois abrir sua App. E isso quando a pla-

taforma (como o Android) não exige que o usuário pré-approve uma lista críptica de permissões.

A Web é totalmente descomplicada. Abra o link no navegador e pronto. E, se quiser, o usuário ainda pode adicionar um favorito em sua tela inicial pra voltar depois no site. Permissões pra coisas mais avançadas são pedidas pelo navegador conforme o necessário (como geolocalização).

Quando há atualização, uma App precisa ser instalada novamente pela loja (seja automaticamente ou manualmente). Em muitos casos, isso envolve baixar a App toda de novo, o que pode ser grande (o Android consegue baixar só as diferenças). Na Web, não precisamos fazer nada. O usuário sempre acessa a versão mais recente quando navega.

Ainda sobre o tamanho da instalação: uma App precisa baixar tudo durante a instalação, todo código e arquivos de todas as funcionalidades. Uma página Web pode carregar só o que é necessário pra tela atual. Se o usuário nunca usar uma tela avançada, ela nem é carregada no browser, mas seria baixada junto com a App, gastando banda. E, por falar em gasto de banda, não há coisa pior do que estar na rua, no 3G, e precisar instalar uma App. A Web costuma ser bem mais leve pra esse tipo de uso rápido e casual.

O ponto principal é que **a Web é totalmente descentralizada**. Você não fica na mão do fabricante que faz exigências pra sua aplicação ser aprovada, proíbe certos usos legítimos, cobra uma porcentagem cara na venda e controla arbitrariamente a exposição da sua App na busca da loja. Na Web, você divulga seu link em qualquer canal, sem depender de ninguém.

Há quem diga que a presença na loja aumenta a exposição da sua App, já que os usuários buscam coisas por lá. É verdade, mas usuários também buscam no Google. E há estudos que mostram que as lojas de aplicativos estão tão cheias hoje que a chance do usuário cair na sua App é bem pequena. Mas, claro, depende da App e do tipo de busca do usuário. Um serviço famoso que tenha marca forte não tem problemas para ser encontrado (buscar Gmail na loja sempre vai trazer a App do Gmail). Mas não há muito espaço para *descoberta* de novas apps. O usuário não vai digitar ‘notícias’ na busca, de tanta porcaria que vai aparecer; e, mesmo que busque, sua Appzinha não vai aparecer no topo.

As Apps de mais sucesso fazem sucesso fora das lojas, e são linkadas em artigos e blogs famosos. As pessoas baixam porque conhecem e já ouviram falar. Raramente baixam porque esbarraram nela na loja sem nunca ter ouvido falar. Então, pensando bem, se você fosse pra Web ao invés de App, daria na mesma.

Outro ponto a se pensar é o uso casual. Uma App exige um compromisso maior do usuário, uma instalação. E muita gente não precisa e nem está disposta a ir tão longe. Muitos usuários só querem uma certa informação num dado momento, algo facilmente encontrável na Web. Uma App é para uso regular, para usuários fiéis. E, mesmo assim, estudos mostram que a maioria das Apps que as pessoas instalam são abertas bem poucas vezes, quando são.

Monetização

Uma diferença brutal nessa parte de distribuição onde as Apps ainda ganham de longe é na questão da **monetização**. As lojas já são plataformas de pagamento integradas e o usuário não tem trabalho algum para comprar Apps e assinaturas. A Web não tem esse tipo de facilidade.

Há os serviços de pagamento como PayPal, mas nada tão fácil. Claro que há a contrapartida da porcentagem altíssima cobrada pelas lojas, coisa que, na Web, você poderia escolher entre diversos serviços de pagamento ou até cobrar diretamente.

Muitas empresas que têm nome mais forte começaram a desafiar esse modelo de loja e retiraram suas Apps das lojas e focaram na Web com pagamento direto pra eles — o caso mais notório foi do jornal americano *Financial Times* (<http://gigaom.com/2011/09/22/financial-times-finds-life-outside-the-app-store-pretty-good-so-far/>).

Mas, tirando essa questão da monetização, o processo de distribuição e instalação é muito mais complicado com Apps que na Web.

Multiplataforma

O grande apelo da Web é ser independente de plataforma. A gente encontra navegador Web em tudo que é aparelho hoje em dia, não importando tipo, marca, sistema operacional. Desenvolver seguindo os padrões Web garante o acesso a todos os usuários do mundo, sem discriminação.

Claro que isso não quer dizer que não existam suas dificuldades. As incompatibilidades entre navegadores têm diminuído bastante nos últimos anos, mas ainda são um problema. É preciso testar bastante e escapar dos bugs. Mas é um trabalho muito menos complicado que fazer Apps para plataformas específicas.

O mais comum, aliás, é que o desenvolvimento de Apps foque nas plataformas que estão em alta — no mundo mobile, isso significa iOS e Android. Sob argumento de que não vale a pena economicamente suportar plataformas menores, muita gente deixa de lado os zilhões de outras plataformas — no mundo móvel, temos Windows

Phone, Blackberry, Symbian, Bada, Tizen, Firefox OS, só pra citar as mais ouvidas. Não é à toa, aliás, que a maior parte desses sistemas não tão famosos usa tecnologias Web (HTML, CSS e JS) como base pro desenvolvimento de Apps. Elas estariam perdidas se exigissem uma reescrita total da App.

Aliás, esse cenário de se usar HTML, CSS e JavaScript para escrever Apps é cada vez mais comum. A ideia é justo resolver o problema da portabilidade — o mesmo código rodaria no Android, iOS, Windows Phone etc, usando sistemas de empacotamento como PhoneGap. Vamos discutir melhor esse cenário daqui a pouco, no tópico 4, mas não se engane: Apps HTML5 compartilham a maior parte dos problemas das Apps de modo geral e não são Web, apesar de usarem a mesma linguagem.

É possível, portanto, ter um mínimo de portabilidade de código ao escrever Apps, mas você ainda vai ter que publicar em lojas diferentes e individualmente. E alguma hora você vai desistir de publicar numa loja menos conhecida pra focar nas maiores (já pensou em publicar na loja do Bada alguma vez?). E assim você vai excluir uma parcela do mercado e discriminar esses usuários. É por isso que falamos que a Web é mais democrática, aberta e acessível.

Experiência do usuário

Há diferenças fundamentais entre Apps e Web sob a ótica da experiência do usuário e de usabilidade. E, na minha opinião, esse acaba sendo o fator mais determinante na hora de escolher qual estratégia seguir.

Argumentos comuns que as pessoas levantam, como performance e acesso a hardware, pra mim são irrelevantes na maioria dos cenários. Poucos casos exigem performance altíssima ou acesso a tudo que é hardware do aparelho. A maioria dos projetos é mais simples que isso e funcionaria bem tanto na Web quanto em App. Mas a diferença na experiência do usuário é algo sempre presente.

A Web tem várias vantagens de usabilidade em relação a Apps. O usuário pode selecionar o texto facilmente, pode dar zoom à vontade (mais sobre isso no tópico 12), pode criar favoritos e atalhos para páginas específicas, pode clicar em links e navegar à vontade no navegador. Apps costumam ser mais limitadas nesses pontos — não posso favoritar uma tela, compartilhar links com as pessoas, dar zoom livremente.

Mas, mesmo com tudo isso, a questão principal é que a **experiência de uso de uma App é diferente da de um site**. Como usuário, você sabe quando está usando uma App ou navegando num site. Ambos podem ter o mesmo conteúdo e prover as mesmas funcionalidades, mas a experiência de uso ainda assim é diferente.

É importante conhecer bem o escopo do projeto e as expectativas do seu grupo

de usuários. Às vezes, por mais que Web faça mais sentido, uma App vai trazer a experiência final mais apropriada naquele caso. Ou não, muitas vezes as restrições das Apps com relação à Web são fatores mais determinantes.

O importante é analisar bem caso a caso e tomar boas decisões. Não adote uma App só por causa de modinha. E não adote a Web só porque é o caminho mais curto e fácil.

CAPÍTULO 4

HTML 5 é diferente de HTML 5: o caso das packaged apps

Você usa HTML, CSS e JavaScript pra escrever suas páginas Web, seus sites. Durante anos, toda frase com ‘HTML’ tinha ‘Web’ no meio. HTML (e seus amigos CSS e JS) é a língua da Web. Mas, nessa revolução mobile, é comum ouvir ‘HTML’ (e mais especificamente ‘HTML 5’) na hora de descrever Apps.

A App do LinkedIn no iOS é feita em HTML 5. O Facebook tinha uma App HTML 5 e falou que não gostava — por isso, migrou pra nativo. A Microsoft fala pra você escrever Apps pro Windows 8 em HTML 5. O Firefox OS é um sistema operacional móvel inteiro construído em HTML 5.

Isso quer dizer que Windows 8, Firefox OS e os outros casos rodam na Web? Não.

O que é Web

Parece ridículo escrever sobre *o que é Web* para o público desse livro. Mas não,

a confusão é grande com essa modinha de ‘HTML 5’. Precisamos definir bem, justo pra entender o que *não é Web*.

Web não é só usar HTML 5, CSS e JavaScript — até porque podemos fazer sites em outras linguagens, como Flash, embora seja cada vez menos comum. Web é a soma de uma rede em cima de HTTP com a capacidade de se criar links entre páginas, em um formato padronizado, o HTML. Web é estar conectado na Internet e carregar suas páginas de um servidor em qualquer lugar do mundo. E, seguindo links, navegar pra todos os cantos.

Que fique claro: HTML 5 é só a *linguagem* usada na Web. Não é a Web.

Reaproveitamento de linguagens

Com a Web explodindo em popularidade, o mundo se viu infestado de programadores HTML, CSS e JavaScript. Mas nem só de Web é feita a programação. Por isso, existem diversas linguagens focados em servidores, programas Desktop, aparelhos embarcados etc.

Até que caiu a ficha de muita gente: se já aprendemos as linguagens HTML, CSS e JS por causa da Web, será que não seria melhor usar essas mesmas linguagens em outras situações e evitar ter que aprender novas? Excelente ideia, apesar da confusão que isso criou.

Se alguém fala que é um *programador JavaScript*, você não pode simplesmente inferir que ele manja tudo de jQuery. Pode ser que ele nem programe o *front-end*, mas foque em programas no servidor usando Node.js. É preciso, então, diferenciar bem os usos que são feitos dessas linguagens hoje em dia.

HTML, CSS e JavaScript em todo lugar

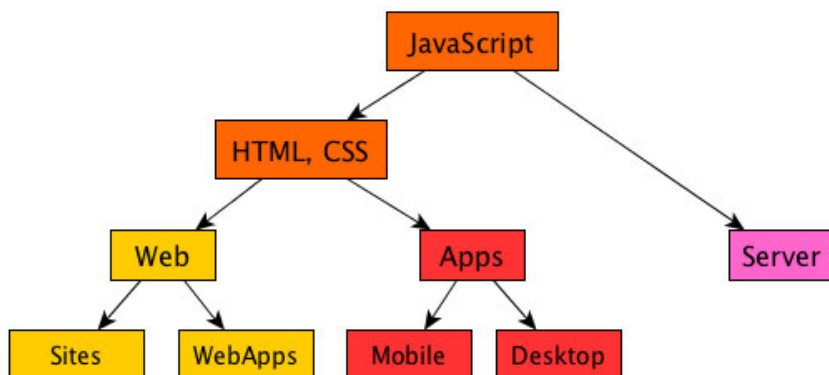
O **JavaScript** é usado comumente hoje em duas situações:

- Em companhia do HTML, CSS, renderizando num **navegador** ou equivalente;
- No **servidor**, como linguagem de programação de uso geral em plataformas como Node.js ou Rhino.

Seguindo o caminho onde usamos JavaScript junto com **HTML e CSS**, que é o que nos interessa neste livro, temos mais duas divisões onde podemos usar esse trio de linguagens:

- Na **Web**, rodando no navegador do cliente após um acesso no servidor. Podemos falar de um *Site* comum ou uma *WebApp* mais complicadinha;
- Em **Apps** empacotadas (*packaged apps*), seja em *mobile* ou *desktop*.

Resumindo, graficamente:



Packaged Apps

A grande novidade do burburinho que se faz em torno de HTML 5 hoje em dia são essas *packaged apps*. Você programa a aplicação em HTML, CSS e JavaScript mas em arquivos locais na sua máquina, sem subir para um servidor. Aí você empacota tudo isso numa App que depois será executada no aparelho como uma aplicação normal.

Por baixo dos panos, a plataforma de execução vai usar uma espécie de browser pra ler esses arquivos e montar a tela. É o que o pessoal chama de **WebView**, uma forma de embutir só o mecanismo de execução do browser sem a parafernália toda do navegador completo — barra de endereços, configurações do usuário, favoritos etc. É, basicamente, só um **renderizador de HTML**.

Essas *packaged apps* estão em todo lugar hoje em dia. O *PhoneGap* é uma ferramenta de empacotamento que gera Apps para várias plataformas a partir de um código HTML comum (ele suporta iOS, Android, Blackberry, Windows Phone, Bada, Symbian e WebOS). Parece simples, mas não é. É preciso gerar um binário para cada plataforma que você tiver interesse e, depois, publicar na loja específica de cada uma.

Outras plataformas já estão sendo montadas com a ideia de *packaged apps* embutida diretamente. O Firefox OS é um exemplo disso. Você escreve tudo em HTML, cria um arquivo de configuração simples e gera um ZIP pronto pra instalar. Windows Phone 8, Tizen e Blackberry 10 são outros exemplos de suporte direto a *packaged apps em HTML*.

No Desktop, essa ideia já existe há anos também. As extensões que instalamos nos navegadores como Firefox e Chrome são escritas em HTML e empacotadas como extensão (e depois publicadas nas respectivas lojas). O Chrome tem também as *Chrome Apps* que são, de novo, aplicações HTML empacotadas e distribuídas na *Chrome Web Store*. O novo Windows 8 fez um estardalhaço como primeiro grande sistema operacional Desktop a suportar desenvolvimento de Apps nativas em HTML, CSS e JavaScript. Mas muitos outros seguem essa ideia — como o Chrome OS e o Gnome no Linux.

No mundo mobile, outro termo comum de se ouvir para essas Apps HTML 5 empacotadas é **Hybrid Apps** (*aplicações híbridas*). E, claro, por ser um HTML rodando numa espécie de browser, você também pode carregar páginas externas do seu servidor dentro da App. Nem todo código HTML precisa estar empacotado na App original.

HTML 5 é diferente de HTML 5

É importante diferenciar o que é usar HTML 5 (e CSS e JS) na Web do que é usar HTML 5 pra escrever *packaged apps*. As Apps, mesmo escritas em HTML, têm todas as características (e limitações) de qualquer App, como discutimos antes. O fato de se usar HTML pra escrevê-las só alivia o fato de ser uma linguagem comum e o desenvolvedor não precisar aprender uma linguagem específica da plataforma — Objective-C no iOS, Java no Android, C# no Windows Phone etc.

Usar HTML 5 como base para Apps também traz um certo nível de portabilidade e facilidade para suportar múltiplas plataformas. Mas cuidado que isso engana. Ao escrever uma App em HTML 5 pro Windows Phone, por exemplo, você vai usar as linguagens comuns mas **muitas APIs específicas** da Microsoft. Mesma coisa se você for criar uma App pro Firefox OS ou pro PhoneGap. A linguagem é comum e muitas APIs mais simples são as mesmas, mas coisas mais complicadas são específicas de plataforma. E isso, claro, limita a portabilidade.

Já quando falamos de Web, falamos de uma **especificação comum** guiada pelo W3C e pelos navegadores. Falamos de **padronização** de APIs e linguagens. Isso sim é portabilidade total. Mas, claro, sabemos que apesar da mega evolução, o HTML

5 não traz todos os recursos das plataformas nativas. Por isso que existem as APIs específicas de cada fabricante, mesmo quando sua App é escrita em HTML, CSS e JS.

O HTML 5 está evoluindo e muitas das inovações proprietárias das plataformas específicas acabam sendo especificadas e se tornando padrão. É um processo mais lento, mas que evolui a Web por igual e garante sua independência e compatibilidade.

CAPÍTULO 5

Design Responsivo por uma Web única

A comunidade de desenvolvedores Web compartilhou durante muito tempo um gigantesco **delírio coletivo**. Uma **ilusão** absolutamente falsa mas que se impregnou de tal forma que todo mundo achava que era verdade. Eu mesmo participei dessa **alucinação coletiva** durante muitos anos.

Estou falando da **ilusão de que a Web tem tamanho fixo**.

Falar hoje que a Web é uma mídia flexível e adaptável soa trivial. Mas, durante anos, o comum na Web eram as páginas de tamanhos fixos. Discutíamos qual era o melhor tamanho pra fazer as páginas. Seria 960px de largura? 980px? Lembro até hoje da mudança nos sites quando a resolução mais comum dos monitores deixou de ser 800px e passou a ser 1024px.

Mas, sempre tivemos monitores de 800px de largura, assim como 1024px. Hoje, as resoluções de 1280px e 1366px são as mais comuns. Mas muita gente usa resoluções maiores, com 1600px ou 1920px. Nossos sites fixos nunca levaram isso em conta.

Programávamos pra resolução mais comum e esquecíamos o resto. Nossos designs eram *pixel perfect* — copiados pixel a pixel do desenho original do Photoshop.

Isso é ridículo. Isso não é a Web portátil, acessível e universal que gostaríamos.

A Web é flexível

Design responsivo nada mais é que (re)lembrar que **a Web é uma mídia flexível e adaptável**. O vergonhoso é pensar que a Web sempre foi assim, nós que colocamos as restrições em nossas páginas.

Muita gente fala disso há tanto tempo: o seminal artigo **A Dao of Web Design**, de John Allsopp, publicado em 2000 no *A List Apart* (<http://alistapart.com/article/dao>), é leitura obrigatória. No início do milênio, ele já falava:

“O controle que os designers conhecem na mídia impressa, e constantemente desejam na web, é simplesmente em função da limitação da página impressa. Devemos abraçar o fato de que a Web não tem as mesmas restrições, e projetar para essa flexibilidade.”

Ou seja, a mídia impressa que é limitada, a Web não. Tão óbvio.

Esse mesmo artigo ainda falava:

“Faça páginas que são acessíveis, independentemente de navegador, plataforma ou tela que seu leitor escolha ou tenha que usar para acessar suas páginas. Isso significa páginas que são legíveis independentemente da resolução ou tamanho da tela, ou do número de cores.”

É quase impossível imaginar que ele não estava falando das novas páginas responsivas que abrem tanto no celular quanto no Desktop. Não, ele escreveu isso em 2000, clamando que focássemos em flexibilidade nas nossas páginas, pois a Web é e sempre foi assim.

Demorou uma década pra cair a ficha. Mas, antes tarde do que nunca.

Design Responsivo

Você certamente já viu diversos sites responsivos por aí. São aquelas páginas que se adaptam a todo tipo de dispositivo. Você abre a mesma página no celular, no tablet e no Desktop, e ela se adapta pra melhorar a experiência do usuário. Vamos explorar depois todos os aspectos técnicos dessas soluções aqui no livro, mas foquemos agora nos aspectos estratégicos.

O termo *Web Design Responsivo* surgiu num artigo divisor de águas publicado em 2010 por *Ethan Marcotte* no *A List Apart* (<http://alistapart.com/article/>

[responsive-web-design](#)) — não por acaso, a frase de abertura é uma citação do artigo *A Dao of Web Design* que vimos antes.

Pouco tempo depois, o próprio Ethan publicou um livro aprofundando as ideias — tenho um review sobre ele no meu blog: <http://sergiolopes.org/review-responsive-design-ethan-marcotte/>

A ideia pra palavra ‘responsivo’ veio da arquitetura, na qual se fala de técnicas para construções e materiais de adaptarem ao ambiente e às pessoas que interagem com ele. Aí o Ethan fala:

“Ao invés de criar designs desconectados para cada um do crescente número de dispositivos web, nós podemos tratá-los como faces da mesma experiência. Podemos criar para uma experiência de visualização ideal, mas embutir tecnologias padronizadas nos nossos designs para fazê-los não apenas mais flexíveis, mas mais adaptados para a mídia que os renderiza.”

A chave pro design responsivo é fazer um **design flexível e adaptável**, que se ajuste às características do navegador, do dispositivo e do contexto do usuário.

Os pilares técnicos das soluções responsivas são o **layout fluído** (tópico 8), o uso de **media queries** (tópico 9) e de **imagens flexíveis** (tópico 19). Vamos lidar detalhadamente com cada um desses pontos nos tópicos seguintes do livro.

O conteúdo é o que importa

Há quem diga que é impraticável oferecer o mesmo site para todo mundo porque, quando navegamos em telas pequenas, *queremos conteúdo mais focado e páginas mais simples e diretas*. É absolutamente verdade que usuários mobile têm menos paciência para páginas grandes e carregadas, mas isso não inviabiliza um design responsivo.

A chave é a **priorização de conteúdo**. É preciso repensar todo o conteúdo para descobrir o que realmente importa, e remover todo o excesso. Uma página mobile não deve ser apenas um design menor, mas uma *completa reestruturação de conteúdo*. Por isso que só adaptar o design de um site Desktop já existente dificilmente funciona.

E, se você já gastou um bom tempo *cortando coisas secundárias e priorizando o importante*, por que oferecer essa versão mais simples e funcional apenas para usuários mobile? Por que temos mania de pensar que usuários Desktop devem receber páginas cheias de informações inúteis?

Responsive design é entregar a mesma informação — útil e priorizada — para todo mundo!

Há muita confusão entre o *meio de acesso* e o *contexto de acesso*. Muita gente acredita que, só porque o usuário está no smartphone, significa que ele está na rua com pressa. Ou que, só porque o usuário está num browser Desktop, ele está sentadinho na mesa dele com todo tempo e paciência do mundo. Usa-se isso para justificar diferenças brutais entre a versão Desktop e a versão mobile de um site. Usuário Desktop supostamente quer mais conteúdo e mais opções; enquanto usuários mobile supostamente querem páginas mais simples e focadas apenas em atividades chave.

Nada disso é verdade.

Um usuário Desktop pode estar num notebook no meio da rua, usando 3G e com pressa. E um usuário de smartphone pode estar sentadinho no sofá de casa usando seu Wi-Fi. Ou vice-versa. Não dá pra saber. Fato é que **o contexto de acesso do usuário não tem nada a ver com o dispositivo que ele está usando**. Não devemos julgar o usuário por seu aparelho.

Na prática, como você nunca vai saber o real contexto de uso, sempre ofereça acesso a *todas* as informações e funcionalidades do seu site para todos os usuários e da maneira mais limpa e clara o possível. Claro que a forma de acesso muda, o design muda (vamos falar mais disso depois), mas tudo deve ser acessível sempre. Jamais corte conteúdo achando que o usuário mobile não vai querer ver aquilo. Otimizar para mobile não significa diminuir funcionalidades, significa focar.

A Web Única

Criar um site responsivo não é fácil. Trabalhar com flexibilidade e adaptação é bem mais complicado que um site fixo em pixels. As ferramentas de desenho ainda não estão preparadas e os designers gráficos costumam ter dificuldades para criar com responsividade em mente.

O código fica mais complexo também. Há dificuldades para se trabalhar com imagens e vídeos responsivos. É bastante complicado adaptar um site Desktop já existente para ser responsivo.

Apesar de tudo isso, **designs responsivos são o futuro**. Pelo simples motivo de que a **Web é única** e criar sites separados para cada categoria de dispositivos de hoje e do futuro é impossível.

Eu adoro esse termo, **One Web**, que o W3C defende quando fala de mobile (<http://www.w3.org/TR/mobile-bp/#OneWeb>). É a ideia pura de que não existe Web Mobile e Web Desktop, como falamos já no início do livro. Existe a Web e várias formas de acessá-la. Pensar assim é a única forma de não enlouquecer nesse mundo de cada vez mais dispositivos diferentes com acesso à Web.

E o design responsivo é a forma mais adequada de entregar o mesmo conteúdo pra todo mundo nessa Web única.

Mas eu quero sites diferentes para Desktop e Mobile!

Nenhuma solução é *bala de prata*. Talvez você tenha bons motivos para não encarar a Web com um site único pra todo mundo. Há razões para se preferir separar o site Desktop do site mobile. Seu site Desktop pode ser muito complicado pra adaptar pra mobile, e um site novo pode ser mais eficiente — vários grandes portais têm seguido essa linha.

Isso quer dizer que você deve esquecer *design responsivo*? **Não.**

Ao criar seu site mobile, você vai precisar, primeiro, definir o que é mobile — o que pode não ser uma tarefa fácil, como discutimos antes. Você terá que decidir se vai atacar só smartphones; se tablets recebem a versão Desktop ou a versão mobile; onde traçar a linha que divide smartphones de tablets; dentre muitas outras perguntas.

Mas vamos dizer que você fechou em fazer um site focado só em smartphones. Mesmo assim, você **vai precisar fazê-lo responsivo**. Só pensando nos tamanhos de tela dos smartphones mais comuns, teríamos 320px, 360px, 380px, 480px, 533px, 568px, 640px e, provavelmente, muitos mais. Como fazer um único site suportar tudo isso? Usando *design responsivo*.

Ou seja, mesmo restringindo a audiência apenas para smartphones, a quantidade de dispositivos ainda é gigante, com os mais diversos tamanhos de telas. Você vai precisar de um site flexível e adaptável.

Mais: essa abordagem pode ser bem interessante para o futuro. Você pode não ter condições agora de recriar seu site completamente responsivo para todos os dispositivos. Então, você pode começar criando uma versão para smartphones, no melhor estilo *mobile first*. Essa nova versão vai ter que ser responsiva. Com o tempo, você pode evoluir seu design para incluir tablets também. Depois, quem sabe, já não inclui Desktops e TVs também? Até um ponto em que seu design responsivo evolui para substituir seu antigo site puramente Desktop.

Eu acredito muito e recomendo a abordagem do **design responsivo e da web única**. Mas mesmo que você tenha sites diferentes, faça-o também responsivo.

CAPÍTULO 6

Mobile-first

Convencido do tamanho gigantesco do mercado mobile e de que a Web é uma só pra todo mundo, você decide investir no design responsivo. Ótimo passo. Hora de tomar a próxima decisão: por onde começar?

Em especial, na hora de criar meu site ou webapp, começo pensando primeiro no Desktop ou primeiro nos dispositivos móveis? Vou precisar cobrir tudo, claro, mas por onde começar? E é tão importante assim definir por onde começar?

Mobile é diferente

A maior diferença entre um dispositivo móvel e um Desktop é o espaço disponível para o conteúdo. A área útil da página. Claro que sempre dá pra dar scroll, mas é fato que cabe menos informação no celular que na telona do computador.

Além do tamanho, há outras diferenças, como o contexto de uso do dispositivo. Um celular é muito mais flexível e pode ser usado em muito mais situações que um Desktop: posso estar na rua, no carro, na fila do banco, em casa no sofá, deitado na cama, assistindo TV, usando o banheiro.

Poderíamos falar também da tendência dos dispositivos móveis terem *touch screen*, o que é quase onipresente já. Mas, como vamos discutir no tópico 23, isso não é mais um diferencial, já que notebooks e monitores com touch são cada vez mais comuns.

Mas fato é que há diferenças claras no uso de aparelhos móveis em comparação ao uso do Desktop. Mais ainda: mobile não é só diferente, parece ser também **mais limitado**. Tela menor, rede lenta, hardware mais lento, touch menos preciso que mouse, etc.

Dispositivos móveis trazem mais liberdade de uso mas trazem também novas restrições que não pensávamos antigamente no Desktop. Como lidar com isso?

Abrace as restrições

É muito mais fácil começar seu design tendo em mente as restrições do mobile e depois evoluir para o modo Desktop que é menos limitado. O contrário é bem mais trabalhoso: criar no Desktop sem restrições e tentar limitar depois conforme for adaptando para o mobile.

Em termos práticos: um **design mobile-first** nos obriga a focar mais e a priorizar melhor o conteúdo. Abraçando as restrições do mobile, acabamos chegando em um design mais simples e funcional, já que não há muito espaço para enrolação. Aí quando evoluímos o design para a versão Desktop, o resultado é também uma interface mais focada e todo mundo ganha.

O caminho inverso é bem mais difícil. Pelo Desktop ser bem menos restrito, acabamos enchendo a páginas de coisas sem problemas. Quando tentamos transformar em mobile, o design começa a parecer exagerado e é bem difícil encaixar tudo.

Criar um design mobile-first é focar no principal, e isso não é fácil. É bem mais fácil fazer uma interface cheia de coisas que uma interface simples que atinja o mesmo objetivo. Bons designs, porém, são simples e funcionais; e pensar primeiro no mobile te ajuda a chegar nisso.

E já que você vai conseguir chegar num produto bom e focado no mobile, pode usar o mesmo na versão Desktop. Aliás, não é porque o Desktop tem mais espaço que devemos entupi-lo de coisas. Telas maiores não significam uma vontade do usuário de ver mais tranqueiras. Interfaces simples e focadas são melhores em todas as situações.

Mas claro que isso também não significa desperdiçar o espaço fantástico do Desktop. Queremos um design simples e conteúdo focado, mas também queremos

aproveitar o melhor de cada tipo de aparelho. Vamos discutir ainda algumas ideias nessa linha no tópico 20, sobre carregamento condicional.

Implementação mobile-first

Criado o *design mobile-first*, chegamos na parte de codificar tudo isso. E temos que pensar novamente por onde começar. Agora que você já tem o design focado e simples na mão, tanto faz começar codificando a versão Desktop ou a versão mobile. Certo? Não, codificar *mobile-first* também é bem mais fácil.

Eu fiz um experimento certa vez: codifiquei duas vezes um site inteiro do zero, uma *mobile-first* e outra *desktop-first*. O código *mobile-first* foi bem mais simples e fácil de entender.

O princípio por trás disso é o *progressive enhancement*. É a mesma ideia de abraçar as restrições, mas agora falando de código. Você escreve um código simples e funcional com o maior suporte possível nos navegadores, e vai evoluindo-o para cobrir funcionalidades mais avançadas de navegadores mais modernos. Aliás, vamos falar bastante de *progressive enhancement* ainda no livro — tópicos 14, 17, entre outros.

Um aspecto interessante de começar a codificar pelo mobile é **performance**. Dispositivos móveis têm restrições gigantescas de performance. Rede 3G lenta e pouco confiável, processador mais lento, menos memória e consumo de bateria são alguns pontos a se ter em mente. Então, se você começa a codificar pelo mobile, grandes chances de focar bastante em performance pra conseguir ultrapassar todos esses obstáculos dos aparelhos.

Aí quando chega na versão Desktop, você tem um site ultra rápido! Focar nas limitações de performance dos dispositivos móveis gera um imenso benefício para o usuário Desktop, que ganha um site muito otimizado.

Aliás, já li muitos artigos que criticam o “absurdo” de sites responsivos terem mesmo tamanho e mesmos requests tanto no mobile quanto no Desktop. A aparente má-prática é que você precisa ter algo mais leve no smartphone que no Desktop. Eu não vejo dessa maneira. É tudo relativo: se minha versão mobile é rápida e enxuta, porque não posso oferecer o mesmo para a versão Desktop? Quer dizer, não há problema em si nos tamanhos do mobile e Desktop serem iguais desde que esse tamanho seja pequeno e otimizado pensando primeiro no mobile. O problema é, na verdade, ter a versão Desktop pesada e tomá-la como base no no mobile, só apertando o design.

Mobile-first garante boa performance pra todo mundo.

Capacidades diferentes

É importante também pensar que um *design mobile-first* não é necessariamente só sobre restrições. Dispositivos móveis têm muitas capacidades que Desktops clássicos não têm.

Por serem aparelhos móveis, há toda uma área de possibilidades em localização: oferecer conteúdos localizados, e permitir ao usuário tirar proveito do fato de estar em certo lugar.

Aparelhos móveis também são usados em contextos diferentes. Eles são usados nas mesmas situações que Desktop (em casa, no escritório), mas também em outras situações que o computador dificilmente chegava: na fila do banco, deitado na cama, no restaurante, dentro de uma loja etc.

Todas essas novas capacidades e possibilidades são impulso para inovação. Começar pelo mobile pode despertar novas ideias que o Desktop não traria. E aí você provavelmente vai precisar lidar com essas diferenças também através de algo como *feature detect*, que veremos no tópico 17.

Outros usos do termo

Falamos de *mobile-first* em dois contextos: no design da aplicação e na implementação do código em si. Mas algumas pessoas usam ainda o termo em outros contextos.

Já vi gente falando de uma *cultura mobile-first* ou de *peças mobile-first*. Isso geralmente quer ressaltar o fato de que uma parcela cada vez maior da população usa seu dispositivo móvel como principal meio de acesso à Web e à sua vida digital. Há até um grupo grande de *mobile-only*, pessoas que nem usam Desktop e só têm smartphone e/ou tablet. Aliás, é justo por causa desse público cada vez maior de usuários majoritariamente mobile que não devemos cortar conteúdo na versão mobile do nosso site (como discutimos no tópico 5), senão essas pessoas jamais teriam possibilidade de ver as funcionalidades cortadas.

Outra ocorrência do termo é quando falam de uma *estratégia mobile-first*. Aqui a ideia é falar da estratégia comercial de um negócio, seu foco como empresa. O Instagram, por exemplo, é tido como uma empresa *mobile-first*. Nasceram como uma aplicação iOS para tirar e compartilhar fotos. Depois, veio a versão Android e, bem por último, um site Web bem simples só pra ver as fotos. É uma empresa que nasceu com foco em mobile. Claro que nem toda empresa segue esse caminho — aliás, a maioria não nasceu no mobile e é bem mais ampla que isso.

E o termo em si — *mobile-first* — nasceu em um post em 2009 pelo especialista mobile Luke Wroblewski (<http://www.lukew.com/ff/entry.asp?933>) no qual ele falava mais da ideia de design e foco. Depois, ele mesmo lançou um livro com esse título onde abordou melhor essas ideias com muitos exemplos práticos de usabilidade e design. É um livro muito bom e interessante — você pode ver meu review aqui: <http://sergiolopes.org/review-livro-mobile-first-luke-wroblewski/>

CAPÍTULO 7

Mercado, Browsers, suporte e testes

Escrevi na introdução do livro que não pretendo falar de números específicos do mercado mobile, já que eles ficariam desatualizados rapidamente. O que interessa, claro, é o *seu mercado*, da *sua aplicação*.

Todo projeto Web precisa definir uma política de compatibilidade dos navegadores. Por mais que falemos da Web única e universal, sabemos que há navegadores de toda qualidade espalhados por aí. E, na maioria dos cenários, não é viável financeiramente tentar alcançar o suporte a todos. É preciso deixar alguns de lado.

Mas como traçar essa linha? Como definir os navegadores que recebem suporte oficial?

Você precisa ver os dados do seu público, analisar as estatísticas do site em questão. Para isso, você vai usar alguma ferramenta de coleta de dados como o *Google Analytics*. Observe atentamente as informações reais do *seu* cenário pra tomar as melhores decisões para o seu público.

Se você estiver iniciando um projeto novo, recomendo olhar as estatísticas da região em questão no *Stat Counter* (<http://gs.statcounter.com/>) e definir um grupo

inicial de navegadores para suportar. Aí coloque o projeto o mais rápido possível no ar para já começar a coletar dados reais e ajustar sua política de compatibilidade de acordo.

Como incluir ou excluir um browser específico

Você vai olhar para seus números e vai achar usuários de todo tipo de navegador. Como decidir qual suportar e qual não?

Uma estratégia é colocar um limite numérico. Navegadores com menos de 1% de uso não serão suportados oficialmente. Ou, se for um site de uso mais geral, coloque um número mais baixo, como 0,5%. Mas é preciso ter um certo *feeling* aqui.

Analise o **histórico de uso** — e não só dados pontuais — e tente concluir baseado na **trajetória** daquele navegador específico. Por exemplo: você olha que o Internet Explorer 10 tem 1% de visitantes no seu site. Devo suportá-lo? Claro que sim, a trajetória do IE10 é de ascensão. Ele ainda tem pouca adoção por ser novo, mas está crescendo a cada dia. A história do IE6, claro, seria outra: é só ver que ele vem diminuindo cada vez mais e, obviamente, não vai crescer.

Esse tipo de análise é mais importante ainda quando for levar em conta navegadores móveis. O percentual no Brasil ainda é baixo com relação aos navegadores Desktop. Mas sabemos que é uma trajetória de ascensão. Você vai querer suportar navegadores mobile na maioria dos novos sites.

Mas você nunca vai conseguir suportar todos os navegadores, sobretudo no mobile, onde o número de plataformas e navegadores diferentes é imenso. Escolha baseado no seu público, no trabalho que teria suportando aquele browser específico, e no retorno esperado dele.

A boa notícia é que os aparelhos modernos têm browsers cada vez mais evoluídos, então não é tão trabalhoso assim suportá-los todos. Mas você pode acabar tendo que decidir excluir plataformas mais antigas como Symbian, S40, BlackBerry, Bada e outros sistemas proprietários.

Muita gente ainda usa celulares mais básicos (*feature phones*), geralmente com algum navegador nativo ou o Opera Mini. Costuma ser um grupo grande de pessoas, mas o trabalho em suportá-los é imenso e você talvez tenha que decidir excluí-los. Eu pessoalmente fico bastante triste quando preciso decidir excluir o Opera Mini de algum projeto. É um browser *muito* usado, mas, nos projetos que participei, suportá-lo demandaria um esforço absurdo e injustificável, infelizmente.

É importante apenas lembrar algo: não suportar e testar oficialmente um certo browser não significa excluir aquele grupo completamente. Se você programar di-

reito, usando as melhores práticas do *progressive enhancement*, é possível que sua página seja minimamente usável nesses navegadores mais velhos.

Como testar?

Testar projetos Web é uma parte pouco divertida do projeto, mas essencial. Embora as diferenças entre os navegadores estejam diminuindo bastante ao longo dos anos, você precisa testar em cada um deles. Vai encontrar bugs, problemas de performance e também observar a experiência do usuário naquele browser ou dispositivo específico.

Você já deve estar craque em testes de sites Desktop. É só instalar vários navegadores em versões diferentes na sua máquina e testar. Mas e com dispositivos móveis? Você *precisa* ter vários aparelhos para testes.

Desenvolver para mobile é algo relativamente caro, pois você precisa montar seu laboratório de testes e comprar vários dispositivos. Ou juntar com outros desenvolvedores e amigos e trocar aparelhos para testes. Outra iniciativa comum lá fora (mas ainda fraca no Brasil) são os *device labs comunitários*, com vários aparelhos doados por todo mundo para uso por todos os associados (<http://opendevicelab.com/>).

Uma estratégia que eu uso é comprar aparelhos usados no Mercado Livre e até alguns com pequenos defeitos. Já comprei um iPod Touch pro meu lab com tela rachada mas que funcionava bem (depois acabei trocando a tela por uma de \$15 achada no eBay).

Você vai precisar escolher aparelhos que representem o melhor possível seu público e o que você vai precisar suportar. Isso depende bastante de cada cenário. Mas, de maneira geral, escolha aparelhos de tipos diferentes, plataformas diferentes, versões diferentes, de vários tamanhos de tela e diversos fabricantes. Uma preocupação que acho importante é em ter aparelhos de várias resoluções diferentes e com várias densidades de pixels (*pixel ratio*).

No início de 2013, conforme escrevo o livro, meu laboratório de testes pessoal tem:

- Samsung Galaxy SII de 4,3” com Android 4.0 oficial Samsung e pixel ratio 1.5;
- Samsung Galaxy Note II de 5,5” com Android 4.1 da Samsung e pixel ratio 2;
- Tablet Android 7” com Android 4.2 e pixel ratio 1 (é um Kindle Fire hackeado);
- HTC Google Nexus One de 3,7” com Android 2.3 e pixel ratio 1.5;

- Samsung Galaxy 5 de 2,8” com Android 2.2 oficial Samsung e pixel ratio 0.75;
- iPod Touch 4 de 3,5” com tela retina e iOS 6;
- iPad 2 de 9,7” com iOS 6.1;
- Nokia XpressMusic com Symbian S60 bem velho.

Essa lista é minha, pessoal, com base nos meus projetos, nas minhas limitações e necessidades. Não coloquei pra você copiar, mas para ter uma ideia de como eu me virei. E, mesmo assim, ainda sinto falta de ter um Windows Phone (tenho planos para comprá-lo em breve), um celular com teclado físico e talvez um iPad Mini e outro tablet Android.

Instalando os browsers

No Android, você vai provavelmente instalar vários browsers para testes. O browser padrão do Android e o Chrome Mobile são os mais usados. Mas há algumas pessoas usando Firefox Mobile ou Opera Mobile. O Firefox pode ser importante se o *Firefox OS* realmente decolar, já que seria o mesmo browser dessa versão do Android (podemos economizar na compra de um aparelho!).

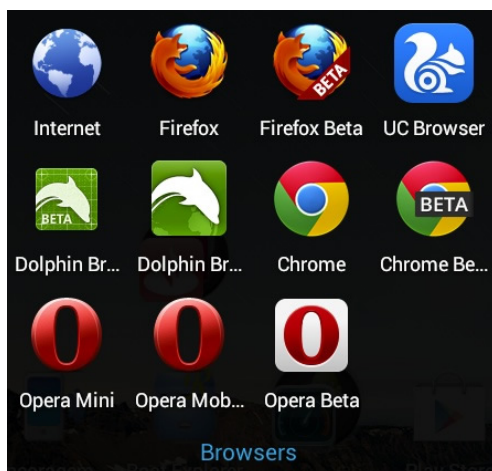


Figura 7.1: Lista dos browsers instalados no meu celular Android, incluindo os beta.

No iOS, o cenário é mais simples já que só o Mobile Safari da Apple é permitido. Até existem outros browsers, como o Chrome, mas ele é na verdade o motor de ren-

derização do Safari com cara de Chrome. A Apple não permite renderizadores de outros browsers — por isso, não temos Chrome de verdade ou um Firefox ou Opera. Há alguns proxy browsers (que renderizam no servidor), como o Opera Mini e o UC, mas você precisa antes ver se vai realmente suportá-los.

No Desktop, você vai ter que instalar algumas versões do Chrome, Firefox e Opera. O Safari é importante, mas só está disponível para Mac. E o Internet Explorer, claro, só no Windows. Então, na prática, você sempre vai precisar de umas máquinas virtuais pra rodar navegadores de outras plataformas (ou use vários computadores).

No caso do Internet Explorer, a boa notícia é que a Microsoft oferece gratuitamente para desenvolvedores Web máquinas virtuais de testes. Eles têm máquinas virtuais prontas pra se rodar no Windows, Mac ou Linux, executando Windows XP, Vista, Windows 7 e Windows 8, com todas as versões do IE desde o 6. É só baixar e rodar (<http://www.modern.ie/virtualization-tools>).

Emuladores para mobile

Como você nunca vai ter condições de comprar todos os aparelhos do mercado, o uso de emuladores pode ser interessante.

Há emuladores excelentes, como o do iOS, com os quais você pode simular iPhones e iPads de todos os tamanhos, rodando iOS desde o 4.x até o mais recente. O ponto negativo é que você precisa de um Mac pra isso. Mas os emuladores em si são muito bons e rápidos de ligar. Você pode testar todo tipo de suporte do Mobile Safari. Única limitação é não conseguir testar as interações touch e a experiência real do usuário com o aparelho. Por isso, é bom ter um outro dispositivo real com iOS também, e só usar os emuladores pra testar os browsers de versões diferentes.

Para Android, há emuladores gratuitos que o Google disponibiliza. Dá pra rodar em qualquer sistema operacional, mas não é a coisa mais fácil de instalar. Eles são mais pesados e demorados para rodar. Como eu tenho 5 aparelhos Android no meu lab, não costumo usar muito os emuladores Android pra testar páginas Web. Mas eles são uma opção caso você precise testar versões diferentes dos aparelhos que você tiver.

O Windows Phone também tem emuladores que você pode instalar. Eles rodam bem em Windows nas máquinas mais novas da Intel. Dá pra rodar no Mac virtualizando, mas ficam bem pesados e exigem uma máquina com pelo menos um Core i5 por restrições nas instruções de virtualização do processador.

Já o Firefox OS é, na verdade, um sistema todo baseado no Firefox. Então, é bem

fácil de testar. Você instala o Firefox normal no Desktop e depois o emulador do OS em cima dele. Ele roda rápido e sem problemas, com a vantagem de estar dentro do browser e te permitir usar Firebug e inspector diretamente.

Há emuladores para outras plataformas também, que você pode pesquisar melhor conforme a necessidade.

Minha dica geral é usar emuladores como um suporte para sua estratégia de desenvolvimento, para não precisar rodar sempre no aparelho. Eu costumo ir desenvolvendo no Desktop, usando emuladores, e só rodo nos aparelhos quando o projeto estiver mais estável. Mas, lembre, é importante sempre rodar em aparelhos reais pra você pegar a experiência real do usuário, sobretudo com relação à performance, à estabilidade e à interação.

Debug em browsers mobile

A maior dificuldade de se desenvolver em mobile é debugar as páginas rodando nos aparelhos. No Desktop, basta instalar uma extensão como *Firebug* ou usar o próprio *Inspector* do navegador (como o *Chrome Developer Tools*). Nos browsers mobile, é bem mais complicado porque geralmente você quer abrir o navegador no aparelho mas debugá-lo no Desktop (não faz muito sentido abrir um debugger na telinha do aparelho).

A mais fácil de todas é o iOS a partir da versão 6 se você tiver um Mac. Basta habilitar a opção de debug no Mobile Safari, conectar o cabo USB no computador e ir no menu *Develop* no Safari do Mac. Lá, você consegue abrir um inspetor associado ao navegador do aparelho (e funciona até com o emulador local).

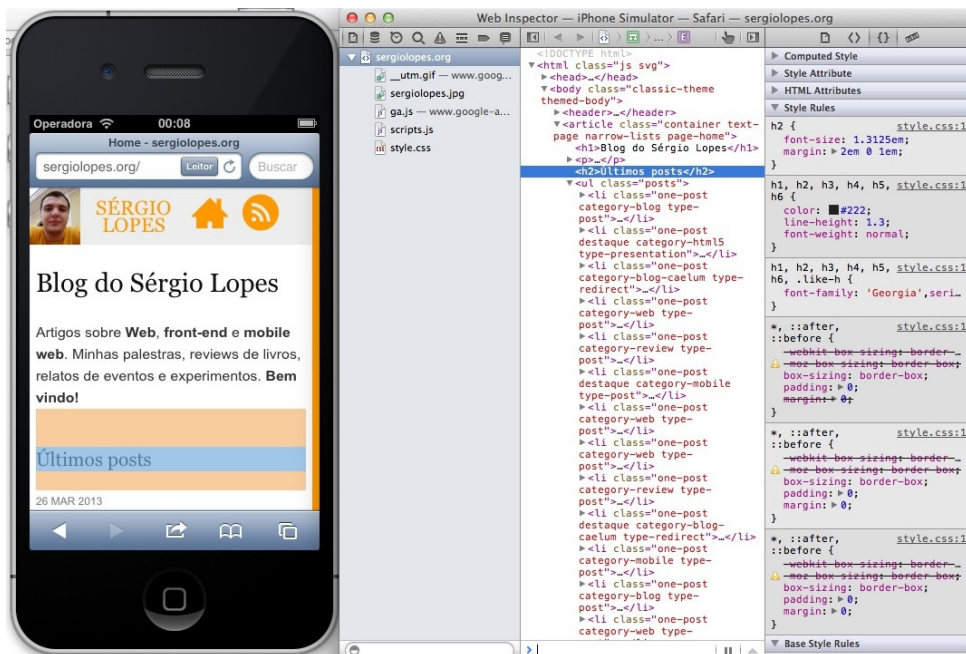


Figura 7.2: Debug do meu blog rodando no emulador do iPhone.

Outros browsers exigem uma configuração mais complicada. Não vou detalhar tudo aqui, já que isso muda com frequência conforme os navegadores vão melhorando. Uns links de referência para documentações de outros browsers:

- Firefox Mobile: <https://hacks.mozilla.org/2012/08/remote-debugging-on-firefox-for-android/>
- Chrome Mobile: <https://developers.google.com/chrome-developer-tools/docs/remote-debugging>
- Opera Mobile: <http://www.opera.com/dragonfly/documentation/remote/>

A maior limitação de debug é no navegador padrão do Android, que ainda não oferece nenhum mecanismo. Mas há soluções espertas para resolver isso. O **Weinre** (<http://bit.ly/apache-weinre>) é um projeto da Apache que permite que pluguemos um debugger remoto em qualquer página rodando no browser mobile (suporta Android, iOS e outros WebKit). O processo de instalação e configuração não é dos mais simples, e o site dele é um pouco assustador, mas a ferramenta é fantástica. Você roda

um servidor no seu Desktop, pluga o debugger no browser do aparelho e abre o Web Inspector no Desktop normalmente.

Para facilitar tudo isso, a Adobe lançou o **Adobe Edge Inspect** (<http://html.adobe.com/edge/inspect/>) que nada mais é que o *Weinre* facilitado. Você instala uma extensão no Chrome no Desktop e uma App no celular e pronto. Ele sincroniza as páginas entre os aparelhos e permite o debug remoto. Porém, é um serviço pago — mas tem uma versão gratuita para testes.

O importante é sempre testar nos aparelhos de verdade. E as ferramentas de debug facilitam bastante o desenvolvimento.

Parte II

Programando a Web moderna



CAPÍTULO 8

Flexibilidade na Web com layouts fluídos

A grande estrela de um web design responsivo é o **layout fluído**. Isso quer dizer não usar medidas fixas como pixels (ou pontos, centímetros, milímetros etc) pra programar o design. Não dá mais pra copiar as medidas no Photoshop da imagem estática que o designer criou com o layout do site. Layout fluído é usar **medidas flexíveis** e é tão velho quanto o HTML em si.

Medidas flexíveis

As duas medidas flexíveis mais usadas são as porcentagens e o `em`. As porcentagens são usadas pra especificar medidas de tamanho com relação ao tamanho do elemento pai.

```
body {  
  /* a página ocupa a largura da tela toda */  
  width: 100%;
```

```
}  
article {  
    /* o article ocupa 3/4 da página */  
    width: 75%;  
  
    /* e tem uma margem interna de 10% do tamanho do pai */  
    padding: 10%;  
}
```

Também podemos usar porcentagens no `font-size`, significando um tamanho de fonte relativo ao tamanho da fonte do elemento pai. O `em` tem esse mesmo significado para fontes — um parágrafo com `font-size` de 200% ou 2`em` dá na mesma, vai ser o dobro do tamanho de fonte do elemento pai.

```
html {  
    /* tem um font-size implícito que equivale  
       a 16px na maioria dos navegadores */  
}  
p {  
    /* parágrafos com tamanho base de 16px * 1.125 = 18px */  
    font-size: 1.125em;  
}  
h1 {  
    /* título principal com o dobro da fonte base */  
    font-size: 2em;  
}  
h2 {  
    /* título secundário é 50% maior que o valor base */  
    font-size: 150%; /* equivalente a 1.5em */  
}
```

O `em` tem a vantagem de poder ser usado em qualquer propriedade mas sempre significar uma relação com o tamanho da fonte. Isso é bem útil quando a medida de algum elemento tem relação com texto, uma medida tipográfica.

Se quiser fazer um espaçamento entre parágrafos, por exemplo, você, provavelmente, vai querer algo relacionado ao tamanho da fonte. Então, um `p { margin: 0 1em; }` faz mais sentido que um `p { margin: 0 5%; }` — com `em`, o espaçamento do texto muda proporcionalmente ao tamanho da fonte.

```
p {  
    font-size: 1.125em;
```

```
/* margin será 18px, já que mudamos o tamanho do em para 1.125em */  
margin: 0 1em;  
}
```

Flexibilidade nos filhos

A grande vantagem das medidas flexíveis é que elas afetam os elementos filhos. Isso ajuda bastante na acessibilidade: o usuário pode aumentar a fonte no navegador dele pra ler melhor e todo o layout baseado em em é afetado. Por exemplo, com fontes:

```
<html>  
<body>  
  <article>  
    <h1>A Web mobile</h1>  
    <p>Não existe Web mobile. Existe A Web.</p>  
  </article>  
</body>  
</html>
```

Vimos que, na maioria dos navegadores, a fonte base mede 16px e é herdada pelos elementos da página. Ou seja, 1em vai ser 16px e 2em vai ser 32px. Mas podemos colocar outros valores:

```
article {  
  font-size: 1.25em;  
}  
h1 {  
  font-size: 2em;  
}  
p {  
  font-size: 0.9em;  
}
```

Qual é o tamanho da fonte de cada elemento?

- O html e o body vão ter os 16px base;
- O article terá $16\text{px} * 1.25 = 20\text{px}$ e esses 20px se tornam a base para todos os filhos de article;
- Assim, o h1 tem $20\text{px} * 2 = 40\text{px}$; e

- O `p` vai ter $20\text{px} * 0.9 = 18\text{px}$.

Um poderoso recurso para aumentar ou diminuir o design proporcionalmente é mudar o `font-size` do elemento pai — por exemplo, mudando o `article` e afetando o `h1` e o `p`.

Nova medida: `rem`

O `em` é bem flexível mas, se tiver muitos níveis de elementos, todos mexendo no `font-size`, é difícil controlar o valor final do `em`. E qualquer mudança num elemento lá no meio afeta seus filhos.

Existe uma outra medida, a **`rem`**, que também é flexível como o `em` mas não é afetada pela hierarquia de elementos pai-filho. O `rem` vem de *root em* e significa um `em` proporcional apenas ao elemento raiz (a tag `<html>`). Na prática, isso significa que `2rem` sempre serão relativos ao tamanho da fonte base do documento (os `16px` iniciais), e não interessa onde o elemento está na página.

Se pegássemos o mesmo código anterior e mudássemos `em` pra `rem`, o resultado seria:

- O `html` e o `body` vão ter os `16px` base;
- O `article` terá $16\text{px} * 1.25 = 20\text{px}$;
- O `h1` tem $16\text{px} * 2 = 32\text{px}$; e
- O `p` vai ter $16\text{px} * 0.9 = 14\text{px}$.

O `rem` ainda é flexível, mas você precisa mudar o `font-size` do elemento `html` pra afetar os valores dos filhos. É útil em algumas situações onde o `em` parece complicar demais. O suporte nos navegadores é bastante bom hoje e a única grande exceção é o IE 8 e anteriores (<http://caniuse.com/rem>).

Existem outras unidades de medidas flexíveis que não citei ainda. Em medidas textuais, existem, além do `em` e do `rem`, a `ch` e a `ex`. Elas são muito específicas e com pouco suporte então não nos interessa muito agora.

Já uma medida nova quem tem ganho bastante destaque são as **`viewport units`**, com `vw` e `vh`. É bem simples: `1vh` é equivalente a 1% da largura da janela do navegador. Com ela, você pode, por exemplo, fazer uma tipografia que escala proporcionalmente ao tamanho do browser, excelente pra design responsivo. Um exemplo:

`h1 { font-size: 3vw; }`. O suporte ainda é pequeno nos navegadores mas tem crescido rapidamente (<http://caniuse.com/viewport-units>). Fique de olho para usar num futuro bem próximo.

Importante são as proporções

Mas você provavelmente já conhecia as medidas flexíveis e porcentagens e em. O difícil é usar as medidas flexíveis de maneira eficaz e sem ficar louco. Pelo menos eu tive bastante dificuldade no início, principalmente depois de anos acostumado a medidas fixas e a copiar os tamanhos direto do Photoshop.

O segredo de um layout com medidas flexíveis é pensar na proporção entre os elementos. É treinar o olho pra enxergar as relações de espaço entre os elementos da página ao invés dos tamanhos fixos do layout. Analise esse design e observe a seção com as 4 notícias:



O primeiro passo seria resistir e não abrir o Photoshop para medir o tamanho de cada bloco. Você tem que olhar para esse desenho e enxergar um painel de notícias dividido em 4 pedaços. Não importa se o design de cada notícia tem 200px ou 300px. Em CSS, estamos falando de algo assim:

```
.noticia-secundaria {
    width: 25%;
    float: left;
}
```

Estabelecemos que as notícias listadas naquele painel de baixo ocuparão um quarto da área disponível. Não interessa o tamanho da tela ou da janela do navegador: a proporção é estabelecida na porcentagem e o tamanho final é recalculado pelo navegador automaticamente.



Figura 8.1: Exemplos desse grid flexível simples pra você testar: <http://sergiolopes.org/mo>

Esse exemplo é bem simples. O difícil é treinar a cabeça a pensar *sempre* em porcentagens e proporções, ainda mais se você vem de anos de web design em pixels, como eu vim.

Nas fontes é a mesma coisa. Você precisa olhar para o design e enxergar que o título principal é 50% maior que o título secundário e estabelecer essa relação no código. Ou seja, uma notícia terá tamanho **1em** e a outra, **1.5em**. Se alterarmos a fonte base da página, todas as medidas em em mudam junto, *proporcionalmente*.

Dica de CSS: box-sizing

Responda rápido: se eu tiver duas `section` com texto, o CSS a seguir vai fazer um layout em 2 colunas de largura igual?

```
section {
    float: left;
    padding: 5%;
    width: 50%;
}
```

A resposta é não. Cada `section` vai acabar com 60% da largura da página e não vão caber uma do lado da outra. A segunda `section` vai escorregar pra baixo da primeira e não vamos conseguir o design em colunas. Por quê? Simples: o padding não é considerado na definição do `width`.

Isso na verdade é o que chamamos de **box model**. É como as propriedades de tamanho de um elemento se relacionam pra determinar o tamanho final do bloco. E o truque aqui é perceber que o `width` não determina a largura do elemento todo, mas apenas do *conteúdo* dele. O tamanho final do elemento é a soma das propriedades `width`, `padding` e `border-width` — e ainda podemos ter uma margem externa com `margin`.

Então, para ocupar metade da tela, o CSS das nossas `sections` deveria ser:

```
section {  
  float: left;  
  padding: 5%;  
  width: 40%;  
}
```

Não sei você, mas eu acho esse *box model* da especificação bastante confuso. É bem mais fácil pensar no tamanho do elemento considerando seu tamanho total, e não excluir padding e border. Bom, mais gente acha isso confuso, a ponto de existir uma propriedade no CSS que troca a forma como o *box model* é calculado.

A propriedade `box-sizing` pode receber o valor `border-box`, significando que o valor da propriedade `width` já considera o padding e o `border-width`. Pare pra pensar um instante e vai ver como isso é bem mais natural. O código anterior que queríamos dividir em duas colunas com padding fica bem mais claro:

```
section {  
  box-sizing: border-box;  
  float: left;  
  padding: 5%;  
  width: 50%;  
}
```

Eu acho o `box-sizing: border-box` extremamente útil em todo tipo de situação e sempre o uso. Mas em design responsivo com medidas flexíveis ele é praticamente obrigatório, de tanto que facilita as contas e deixa os números mais claros.

Um cenário que passo frequentemente é o de usar medidas diferentes em `width` e `padding`. É bem comum que a largura do elemento seja definida como uma porcen-

tagem do elemento pai (como os 50% que colocamos no exemplo). Mas o padding pode ter uma relação maior com o texto e fazer mais sentido ser especificado em em.

Coluna da esquerda, ocupando metade da tela.

Tem texto aqui dentro e as coisas devem se alinhar harmoniosamente.

E essa é a coluna da direita, também com metade da tela.

E também com parágrafos de texto dentro.

Com o box-sizing é bem fácil fazer isso:

```
section {  
  box-sizing: border-box;  
  float: left;  
  padding: 1em;  
  width: 50%;  
}
```

Sem o box-sizing, o código acima não seria possível. Não dá pra colocar um width em porcentagem e um padding em outra medida usando o box model padrão. Pense sobre isso.

O suporte nos navegadores é excelente — até o IE 8 suporta. Alguns navegadores precisam de prefixos. Eu uso esse truque em todos os meus projetos, sempre assim:

```
* {  
  -webkit-box-sizing: border-box;  
  -moz-box-sizing: border-box;  
  box-sizing: border-box;  
}
```

(mais aqui: <http://sergiolopes.org/css-box-sizing-border-box/>)

Restringindo o layout fluído

Fazer um design totalmente fluído é o único jeito de atacar os múltiplos tamanhos de tela dos vários dispositivos diferentes de hoje. Mas, muitas vezes, deixar a página toda ocupar 100% do navegador pode não atingir resultados tão interessantes, principalmente nos extremos, em telas muito grandes ou pequenas.

Imagine o design das 4 notícias que fizemos antes em um monitor gigante com 4000px de resolução. Vai funcionar, vai ficar tudo em 4 colunas, mas com um design bizarro e mega espaçado. É bem comum que queiramos um layout fluido e flexível mas com certos limites em casos específicos.

A propriedade `max-width` do CSS é bem útil nesses casos. Podemos aplicá-la no `body` pra restringir o tamanho da nossa página a, no máximo, um certo tamanho:

```
body {  
    max-width: 2000px;  
    margin-left: auto;  
    margin-right: auto;  
    width: 100%;  
}
```

No caso de um browser gigante, nosso design fixa em 2000px e fica centralizado na tela. Claro que o ideal, pensando em flexibilidade total, seria não ter essa restrição e deixar tudo fluir. Mas o ideal também seria nosso design de adaptar e ficar bonito também nessas resoluções gigantes. E nem sempre é o caso.

Nem sempre vale a pena programar o design pra tantas resoluções diferentes e precisamos focar. Se você fizer o layout totalmente flexível e fluido até 2000px, vai atingir 99% do mercado atual. E as telas mega gigantes ganham um site funcional, limitado a 2000px. É bom pra todo mundo.

Tenha o `max-width` (e o `min-width`) na manga quando precisar colocar certas restrições no layout. Evite, claro, abusar disso. Não vá colocar um `max-width: 800px` sabendo que os Desktops de hoje são bem maiores que isso. Escolha um valor que faça sentido no seu projeto.

O mais importante é que esse `max-width` é uma regrinha a mais apenas com uma restrição final no design. Todo o resto você vai fazer fluido e em porcentagens. Ou seja, se um dia precisar atingir mais resoluções ou quiser ir pro caminho do 100% fluido, basta remover ou trocar seu `max-width`.

Aliás, hoje eu costumo desenvolver usando medidas flexíveis até os sites de *tamanho fixo*. Às vezes, o projeto ou cliente quer um tamanho fixo em 960px, por exemplo. Nem por isso saio usando pixels no CSS todo. Podemos fazer tudo em porcentagens e em, e só colocar um `body { width: 960px; }` no final.

No dia em que o cliente decidir pelo fluido, é só tirar essa restrição do `body`. Além disso, porcentagens podem ser bem mais legíveis que pixels — é bem mais claro dividir essa página de 960px em 5 colunas usando um `width: 20%` do que olhar pra `width: 192px` e enxergar as 5 colunas.

CAPÍTULO 9

Media queries: as melhores companheiras de um layout fluído

Na apresentação do *design responsivo*, o pai da técnica, Ethan Marcotte, falava de 3 componentes: layout fluído, media queries e mídias flexíveis. Muita atenção se dá para as media queries, mas o central é o **layout fluído**, por isso dediquei o tópico anterior inteiro a isso.

Mas, claro, o design fluído não resolve todos os problemas. Aí que as **media queries** entram. Elas são essenciais para um bom design responsivo.

As limitações do layout fluído

Uma página construída com medidas flexíveis é totalmente fluída e já está 90% no caminho de um design adaptável a todo tipo de tela. Mas é difícil fazer tudo ficar bonito e ajustado a todo tipo de resolução só com porcentagens e `em`.

Pegue o design do tópico anterior, que tem 4 notícias divididas com `width: 25%`. O site vai funcionar no celular, mas fica bem estranho dado o pequeno tamanho da

tela:



Repare até que o título da página ficou bom, nem precisa de ajuste: só o uso de medidas flexíveis foi suficiente. Mas as notícias ficaram apertadas e sobrepostas. O ideal seria, em telas pequenas, mostrar uma notícia em cima da outra, com 100% da largura.

Media queries

O golpe de mestre veio com as **media queries do CSS3**. Com elas, adicionamos **design condicional** que só será aplicado em determinada situação. Isso permite re-adaptar o design da página de acordo com características do navegador. No exemplo das notícias, podemos escolher colocá-lo com 100% de largura nas telas pequenas e deixá-lo flutuando com 25% do tamanho nas telas grandes:

```
.noticia {
  float: left;
  width: 25%;
}
```

```
@media (max-width: 400px) {  
  .noticia {  
    float: none;  
    width: 100%;  
  }  
}
```



Figura 9.1: URL do exemplo: <http://sergiolopes.org/m1>

A sintaxe do bloco `@media` recebe uma condição e, dentro dele, as regras de CSS

que só serão aplicadas caso a condição seja válida. No exemplo, o `(max-width: 400px)` indica que só vamos aplicar o CSS em questão quando a janela do navegador tiver, no máximo, 400px. Você pode testar redimensionando o navegador no Desktop ou abrindo no celular (que costuma ter tela com largura de 320 ou 360px).

Media queries são o ponto central da capacidade de adaptação dos designs responsivos. São essenciais em projetos que envolvem dispositivos móveis mas ganharam seu espaço até em soluções puramente Desktop. São uma das principais ferramentas do desenvolvimento Web moderno.

Breakpoints

Mas, claro, é inviável criar uma *media query* para cada tamanho de tela existente no mundo, por isso a importância do layout fluído. Usando medidas fluídas, seu design se adapta naturalmente a várias resoluções e, só quando necessário, as media queries entram, pontualmente, para ajustar o design e melhorar a experiência do usuário.

Esse ponto em que decidimos colocar uma media query é chamado de **breakpoint**. É o ponto de quebra do nosso layout fluído onde uma reestruturação maior é necessária.

No nosso exemplo, podíamos ainda colocar mais uma media query com outro breakpoint intermediário. Já temos o layout em 4 colunas nas telas grandes e em 1 coluna nas telas pequenas. Podemos adicionar um design em 2 colunas num tamanho de tela intermediário:

```
@media (max-width: 600px) {  
  .noticia {  
    width: 50%;  
  }  
}
```



Figura 9.2: URL do exemplo: <http://sergiolopes.org/m2>

Podemos colocar várias media queries, com condições diferentes, no meio do nosso CSS. A ordem, porém, é importante, já que a regra que vem por último sobrescreve a anterior.

Além de `max-width`, podemos usar a `min-width` nas media queries. Essas são, sem dúvida, as media queries mais usadas e as que você mais vai ver na prática. Há algumas outras, porém, que podem ser interessantes pontualmente.

Outras media queries e por que você não vai usá-las

Você pode também colocar uma media query com valor exato ao invés de `max/min`, usando algo como `(width: 320px)`, mas isso é bem inútil na prática. Só vai pegar telas com *tamanho exato* de 320px, restringindo a fluidez do design.

Existe a media query `device-width` (e as versões com `max/min`), que representa o *tamanho da tela do aparelho*. Isso é diferente de usar o `width` normal, que representa o *tamanho da janela do navegador*. Na maioria dos celulares, esses valores são

equivalentes já que o navegador é sempre em tela cheia. Mas existem navegadores mobile que aceitam ser redimensionados, assim como no Desktop. Na maioria dos casos, você não está interessado no tamanho da tela, mas sim no tamanho do navegador.

Para todos os casos que usamos `width` (com `max`, `min`, `device`), também é possível usar `height`. Essa media query pode ser útil em alguns casos, mas 99% das vezes você está mais interessado na largura e não na altura do navegador. Isso porque a Web é uma mídia que flui verticalmente e o comprimento da página não interessa tanto.

Além de controlar tamanhos, é possível usar media queries que representem capacidades e características do navegador. A mais útil é a `orientation` que pega a orientação do aparelho — podemos usar (`orientation: portrait`) ou (`orientation: landscape`). Você pode usá-las para otimizar o design de acordo com a forma que o dispositivo é segurado.

Há outras media queries ainda mais específicas, com uso limitado, e pouco suporte nos navegadores. Não vamos discutir muito aqui, mas você pode consultar: https://developer.mozilla.org/docs/CSS/Media_queries

Na esmagadora maioria dos casos, você vai usar apenas `max-width` e `min-width`.

Operadores nas media queries

Podemos usar a palavra `and` para fazer um E lógico. No exemplo a seguir, a media query só será avaliada se ambas as condições forem verdadeiras ao mesmo tempo:

```
@media (max-width: 600px) and (orientation: portrait) {  
  
}
```

Fazer um OU lógico é equivalente a separar várias media queries por vírgulas, como no CSS normal. No próximo exemplo, o mesmo CSS será aplicado caso alguma das duas condições seja verdadeira:

```
@media (min-width: 300px), (min-height: 300px) {  
  
}
```

É possível também negar uma media query usando a palavra chave `not` no início dela.

A história dos media types

Desde muito tempo o CSS tem suporte para se definir regras que só valem em certo contexto, bem antes do CSS3 e das media queries. Os **media types** permitem que se use estilos diferentes em situações diferentes e sempre foram muito usados para distinguir a renderização na tela da impressão:

```
@media screen {  
    /* CSS para telas */  
}  
  
@media print {  
    /* CSS para impressão */  
}
```

Todos os navegadores modernos suportam esse media type `print` que é aplicado apenas quando vamos imprimir uma página (útil para esconder o menu de navegação ou aumentar a fonte do texto, por exemplo).

Alguns celulares antigos suportavam também o tipo `handheld` para estilos específicos para sites mobile. Os smartphones modernos como iPhone e Android, porém, ignoram o media type `handheld` pois são capazes de renderizar sites completos e não apenas as versões simples feitas para os celulares antigos.

Como então escrever CSS específico para mobile pensando em smartphones e tablets que não se encaixam no antigo media type `handheld`? É aí que entraram as media queries do CSS3, com muito mais recursos e flexibilidade. É possível até misturar tudo e fazer coisas como:

```
@media screen and (min-width: 400px) {  
  
}
```

O código acima funciona perfeitamente nos navegadores com suporte a media queries, mas tem um efeito indesejado nos navegadores antigos. Muitos deles não entendem as media queries do CSS3 mas entendem os antigos media types (como o `screen` usado no exemplo). Na prática, eles acabam avaliando esse código como se fosse só um `@media screen`, ignorando a segunda parte que não entendem.

Isso é ruim, claro, pois a media query será avaliada erradamente nesses navegadores. Pra contornar o problema, as media queries do CSS3 suportam uma palavra-chave nova que não faz absolutamente nada: `only`. Podemos escrever:

```
@media only screen and (min-width: 400px) {  
  
}
```

Essa media query é completamente equivalente à anterior. A única diferença é que a palavra `only` não é reconhecida pelos navegadores antigos e, portanto, eles não vão reconhecer essa media query como verdadeira. É uma gambiarra pra resolver esses problemas de incompatibilidade.

Muita gente usa essas media queries com `only` sem nem entender o porquê. Eu não uso assim. Na maioria da vezes, não quero que minhas media queries sejam aplicadas somente a telas (`screen`), mas a todo tipo de mídia. Quero apenas configurar um certo tamanho, não importando se é tela, TV ou impressão. Recomendo usar apenas:

```
@media (min-width: 400px) {  
  
}
```

Essa media query é mais simples, mais fácil de ler, não tem problemas de ser executada em navegadores antigos e suporta todo tipo de media type. Na maior parte dos cenários, não é necessário colocar o `only screen`.

Conclusão

As media queries são essenciais pra construção de páginas responsivas. Apesar do layout fluído fazer boa parte do trabalho, as media queries é que são responsáveis pela capacidade de adaptação dos layouts. É essencial sabê-las usar bem.

Vamos voltar a falar de media queries em outros tópicos ([13](#), [14](#)) pra discutir algumas boas práticas e ver em mais detalhes como elas ajudam nosso design responsivo. Vamos discutir também como media queries são interessantes até para acessibilidade de sites Desktop (tópico [16](#)).

Mas antes precisamos resolver uma questão. Se testar as media queries como vimos aqui, elas vão funcionar perfeitamente no Desktop e você pode visualizá-las redimensionando a janela do navegador. Mas, ao rodar no smartphone, vai parecer que não estão sendo aplicadas. É que falta discutirmos um outro ponto essencial para páginas mobile, o **viewport**, assunto do próximo tópico ([10](#)).

CAPÍTULO 10

Tudo que você queria saber sobre viewport

Entender como os elementos que você cria na sua página vão ser desenhados na tela é essencial. Muita coisa acontece dentro do navegador e uma delas, importantíssima, é a forma como as medidas são calculadas e exibidas pro usuário. Faz algum tempo que simplesmente colocar um `width: 100px` num elemento não vai necessariamente fazê-lo ocupar 100 pixels na tela.

Um pixel não mede um pixel.

Por mais estranho que pareça agora, o tamanho de um pixel depende de uma série de fatores. E os navegadores dos dispositivos móveis trouxeram ainda mais questões a serem consideradas. Telas pequenas, alta resolução, retina, zoom, viewports. Vamos entender tudo isso.

Um pixel num navegador normal no Desktop

O normal é você criar uma página, dar os tamanhos dos elementos e eles serem

desenhados na tela ocupando exatamente esse tamanho. Coloque um logotipo de 300px de largura e ele ocupará 300px da tela física do computador. Isso é normal. Se colocar algum elemento medido em porcentagem, ele tem seu tamanho calculado em pixels proporcionalmente ao tamanho da janela do navegador.

Tudo isso funciona ok num cenário normal no Desktop. Você pode até descobrir o tamanho da tela do computador usando, em JavaScript, `screen.width` e `screen.height` — vai devolver, digamos, 1280 e 768. Aliás, essa medida é, 99% das vezes, bastante inútil pra desenvolvedores Web. Estamos mais interessados no tamanho do navegador que no tamanho da tela, afinal o navegador pode não estar maximizado.

Você pode pegar o tamanho disponível no navegador com `window.innerWidth` ou ainda com `window.outerWidth` se quiser considerar a janela toda (incluindo barra de ferramentas, barra de endereços etc).

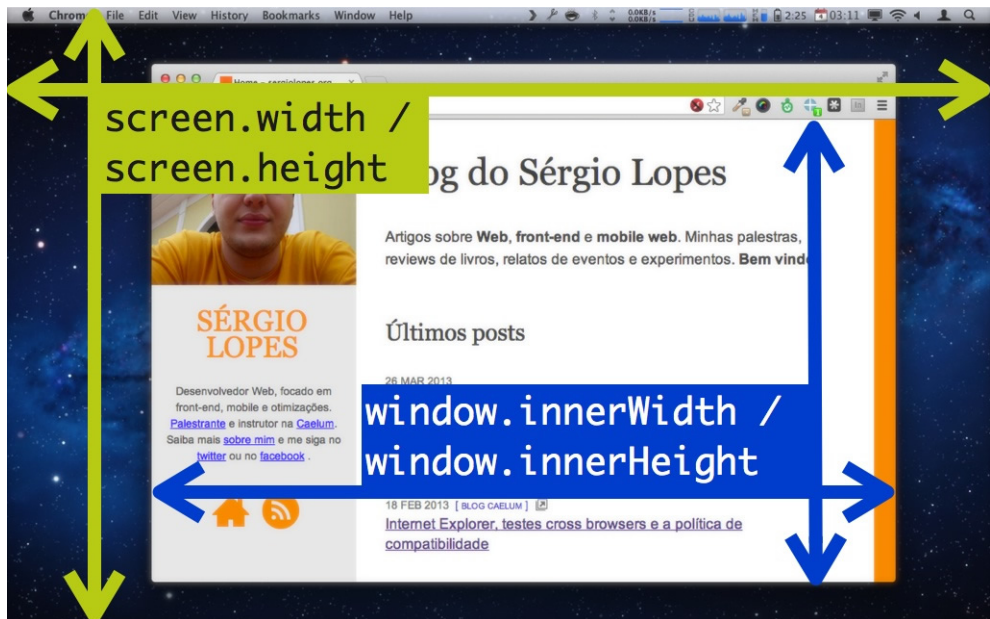


Figura 10.1: Tamanho da tela e do viewport do navegador no Desktop.

O viewport

O espaço disponível para a página ser renderizada no navegador é o que chamamos de **viewport**. Ele depende do tamanho da janela do navegador e desconsidera as

barras de ferramenta, barra de rolagem, navegação etc. É aquele espaço em branco onde a página abre e, como vimos, podemos medi-lo com `window.innerWidth` e `window.innerHeight`.

Num navegador hipotético em tela cheia que não tenha nenhuma barra de ferramentas e só mostre o conteúdo da página, é claro que o tamanho do viewport (`window.innerWidth`) é igual ao tamanho da tela (`screen.width`). Certo? Não, depende.

O zoom de página

Todo navegador moderno permite que o usuário dê zoom na página. É um recurso essencial para acessibilidade — nem sempre o usuário tem visão perfeita, e nem sempre o desenvolvedor deixou as coisas com tamanhos grandes e legíveis. Estamos falando aqui daquele zoom dos navegadores Desktop (esqueça mobile por enquanto), que geralmente você acessa usando `Ctrl + /` `Ctrl -` (ou usando `Cmd` no Mac).

Pense: se você der zoom de 200% na página, o que você espera que aconteça com aquele logotipo que ocupava 300px na tela? Claro que você espera que ele fique com o dobro do tamanho e seja desenhado ocupando 600px da sua tela. Isso que o zoom de página faz, aumenta tudo de tamanho.

Mas e o código da sua página, muda quando o usuário da zoom? Obviamente, não. Você continua escrevendo que a imagem tem `width:300px` mas o navegador sabe que, agora, deve desenhá-la ocupando 600px, já que o zoom foi de 200%. Ou seja, um pixel não é mais um pixel. Você escreve 300px mas ele é desenhado com 600px. Confuso?

A questão aqui é que estamos falando de *pixels diferentes*. Chamamos de **pixel físico** (ou *device pixel*) aquele menor ponto na sua tela que pode receber uma cor. E chamamos de **CSS pixel** aquela medida que você escreve no seu código usando o `px`. E eles são diferentes!

Na verdade, na maioria dos casos, um *pixel físico* é igual a um *CSS pixel*. Mas, quando o usuário dá zoom, as coisas mudam. Um zoom de 200% faz 300 *CSS pixels* serem renderizados como 600 *pixels físicos*. A regra é que **número de pixels físicos = número de CSS pixels × zoom**.

O viewport muda com zoom

Ao dar zoom, as coisas ficam maiores e cabe menos informação na tela. A tela

continua do mesmo tamanho, mas o espaço disponível pra sua página fica menor — o viewport diminui.

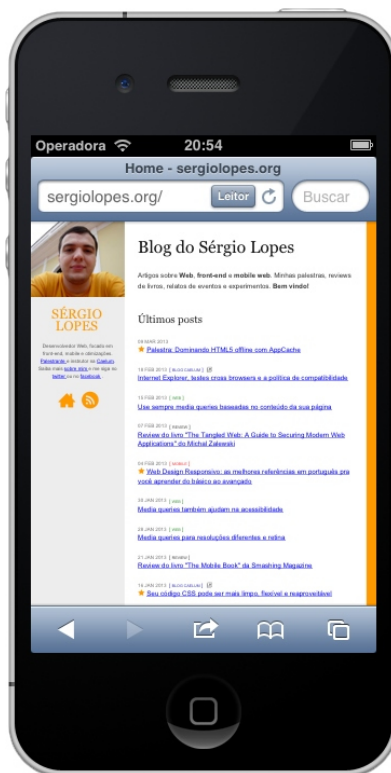
Isso porque o tamanho da tela (`screen.width`) é medido em *pixels físicos*. Mas o tamanho do viewport (`window.innerWidth`) é medido em *CSS pixels*, que mudam conforme o zoom aumenta.

Outra medida importante é o tamanho do conteúdo da página em si, de tudo que está dentro do elemento `<html>`. O viewport é a área visível mas o conteúdo é geralmente bem maior que isso, por isso damos scroll. Você pode medir o tamanho da sua página, em CSS pixels, com `document.documentElement.offsetWidth`.

Depois, faça o teste nessa página abrindo no Desktop e dando zoom: <http://sergiolopes.org/m3>

As telas dos dispositivos móveis

Os smartphones modernos pós-iPhone nasceram com um dilema: têm uma tela relativamente pequena mas um navegador incrível capaz de renderizar todo tipo de página, tanto as otimizadas para dispositivos móveis quanto as feitas pra Desktop. Agora, qual é a mágica para uma página pensada pra um Desktop de 1024px caber na telinha pequena de 320px de um iPhone? A página é **redimensionada** e aparece menor no celular:



A imagem anterior é do meu blog pessoal em sua versão Desktop abrindo num iPhone. Ele abre igualzinho um navegador Desktop, todo ajustado. Mas, claramente, ele não obedece meu CSS, senão minha foto, que foi declarada como `width: 280px` ia aparecer ocupando quase toda a pequena tela de 320px do iPhone.

De novo aqui a ideia do viewport, pixels físicos e CSS pixels. **Um iPhone tem 320 pixels físicos de largura mas assume um viewport de 980 CSS pixels.** Releia essa última frase e esteja certo de entender tudo que ela diz.

Do ponto de vista do código da nossa página, enxergamos 980px. Por isso, minha foto de 280px ocupa mais ou menos 28% da tela, como no screenshot que vimos antes. O que o navegador móvel faz é parecido com a ideia do zoom do Desktop. Ele usa uma escala diferente pro CSS pixel em relação ao pixel físico. Ele vai desenhar 320px na tela, mas o desenvolvedor enxerga os 980px. Todo o redimensionamento é feito pelo navegador.

Zoom mobile e os dois viewports

Claro que o site Desktop aberto no celular fica bem difícil de usar. Ele abre pequenininho mas o usuário depois faz o gesto de pinça (*pinch*) pra fazer um zoom e ver o pedaço da tela que interessa.

Repare, porém, que esse gesto de zoom que fazemos no smartphone é diferente daquele zoom que fazemos no Desktop. Esses modos de zoom são tão diferentes que damos nomes diferentes pra eles. O que fazemos no celular chamamos de **redimensionar página** (*page scale*) e o do Desktop é **zoom de página** (*page zoom*).

O *page scale* no celular só dimensiona o pedaço visível da página, ele não altera o design da página. O site continua renderizado igualzinho e o gesto de pinça só faz a gente *observar* um pedaço específico. É diferente do *page zoom* no Desktop, que aumenta o tamanho dos elementos e renderiza novamente a tela.

Em outras palavras, o **page zoom altera o tamanho do viewport** enquanto o **page scale altera o quanto vemos do viewport, mas ele continua igual**. A página continua com 980px no celular enquanto fazemos o gesto de pinça, só que enxergamos um pedaço menor dela.

Por isso, falamos que, em dispositivos móveis, existem **dois viewports**. Há o viewport que representa a área disponível para nossa página. Esse viewport, que vamos passar a chamar agora de **layout viewport**, mede 980px fixos no iPhone.

O outro é o **visual viewport** que é o tanto que estamos vendo atualmente na tela. Se abrir a página sem dar zoom, o *visual viewport* é do tamanho do *layout viewport* — podemos ver a página toda, como na Figura anterior. Mas, quando fazemos o gesto de *page scale*, diminuimos o *visual viewport* para mostrar só um pedaço do *layout viewport*:

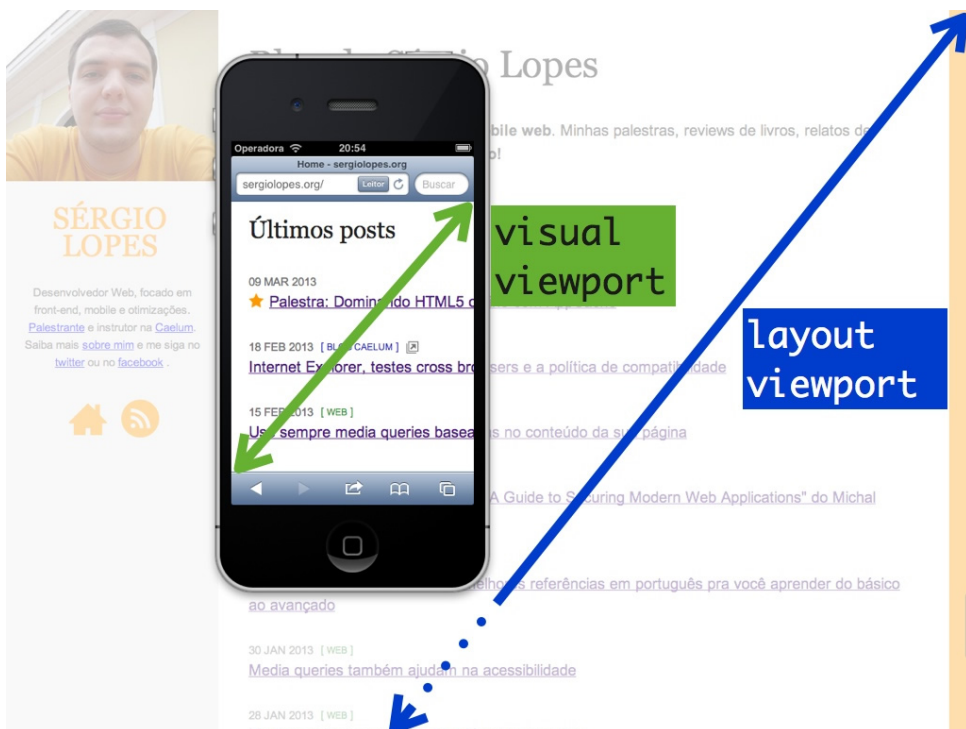


Figura 10.2: Meu blog no iPhone dando um page scale num pedaço da página. O layout viewport é o mesmo mas só uma parte está visível no visual viewport.

Podemos medir o **layout viewport** com `document.documentElement.clientWidth` e o **visual viewport** com `window.innerWidth`.

Esses conceitos são bem complicados. Entender completamente a relação entre os viewports diferentes não é algo fácil, mas é essencial. Rode o teste a seguir em um dispositivo móvel moderno com um navegador atual para ver as diferentes medidas. Faça os gestos de zoom, navegue, gire o aparelho e observe como os viewports se relacionam:



Figura 10.3: URL do exemplo: <http://sergiolopes.org/m3>

Geralmente, não estamos interessados no tamanho do *visual viewport*. Lembre que as medidas da página são sempre relativas ao *layout viewport*.

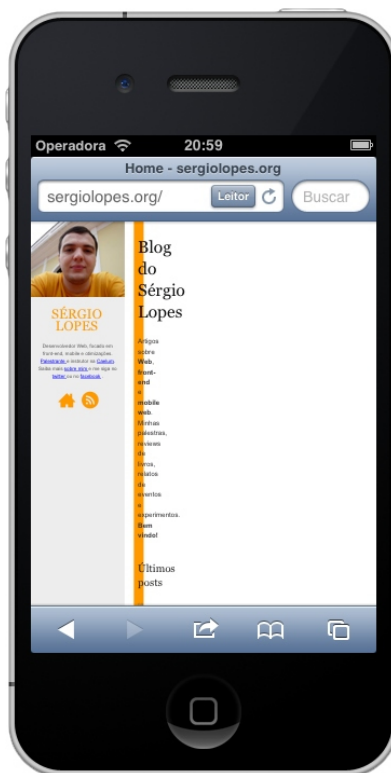
Ajustando o layout viewport

O layout viewport padrão no iPhone mede 980px de largura, pois assume que seu site foi pensado para Desktop. O navegador do Android pensa parecido e usa um viewport padrão de 800px. Já o Opera Mobile usa 850px e o Internet Explorer no Windows Phone, 974px.

Mas abrir um site Desktop no celular é uma experiência pouco agradável. Frequentemente, vamos querer criar uma página otimizada para mobile, que não demande tanto zoom e já mostre o conteúdo em tamanho e formato interessantes para uma tela tão pequena.

Como fazer? Obviamente, não podemos deixar a página com layout fixo em, por exemplo, 960px. Podemos tentar porcentagens e colocar um `width:100` no elemento principal, pensando em se adaptar a diversos tamanhos de tela. Mas isso não vai resolver: o *layout viewport* é grande (980px no iPhone) e os 100% no CSS vão significar tudo isso. O site é mostrado como se fosse de Desktop, com zoom mínimo e conteúdo praticamente ilegível.

Que tal colocar `width:320px`, o tamanho real do dispositivo?



O *layout viewport* continua em 980px mas o conteúdo fica, apertado, em 320px. O usuário precisa dar zoom para visualizar e, pior, a página fica quebrada e com um imenso espaço em branco.

O que precisamos é uma forma de **redimensionar o layout viewport** para que ele seja mais adequado à tela pequena do mobile. A Apple introduziu uma **meta tag viewport** no iPhone que, depois, foi adotada em praticamente todas as plataformas móveis - Android, Opera, Windows Phone etc.

```
<meta name="viewport" content="width=320">
```

Isso indica ao navegador que o *layout viewport* deve ser 320px. Agora, colocar `width:100%` vai significar 320px, deixando a visualização mais confortável.



Refaça os testes de medição do viewport agora que ajustamos a meta tag:



Figura 10.4: URL do exemplo: <http://sergiolopes.org/m4>

Viewport flexível com device-width

Deixar '320' fixo na nossa tag de viewport pode não ser uma boa ideia. Há diversos aparelhos diferentes no mercado, cada um com um tamanho específico. Por

exemplo: um Galaxy S3 tem 360px de largura, um iPad tem 768px, um Nexus 4 tem 384px, um Galaxy Note tem 400px, um Kindle Fire tem 600px e assim por diante.

É possível deixar a **meta tag viewport com tamanho flexível**, baseado no tamanho do aparelho. Basta usarmos:

```
<meta name="viewport" content="width=device-width">
```

Isso assumirá o valor específico pra cada aparelho, que é o valor mais adequado para seu tamanho, conforme determinado pelo fabricante.

A meta tag viewport pode receber alguns outros parâmetros. O mais comum e recomendado para a maioria dos sites mobile é:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

O parâmetro `initial-scale=1` indica para o navegador que a página deve abrir no tamanho especificado (você poderia dizer pra página abrir já com um zoom inicial). Esse parâmetro não parece muito útil, mas ele é importante no iOS. Sem ele, o Mobile Safari considera a largura da página sempre igual ao tamanho no modo retrato, mesmo com o aparelho virado em paisagem. Ou seja, num iPhone, a página teria 320px tanto em retrato quanto paisagem. Coloque o `initial-scale=1` e isso não acontece, o tamanho certo é usado em ambas as orientações.

Agora, configurando o viewport com `device-width`, um CSS pixel é do tamanho de um pixel físico. Bem, pelo menos nas telas comuns. As telas de alta resolução e retina mudam um pouco tudo isso. É o próximo tópico (11).

Viewport em CSS

A parte mais irônica de toda essa discussão sobre viewport é que usar a meta tag não é maneira oficial de se fazer. A meta tag viewport foi inventada pela Apple e copiada por todo mundo. Acabou virando um padrão de mercado. Mas, oficialmente, há outra especificação que lida disso no W3C, a *CSS Device Adaptation* (<http://dev.w3.org/csswg/css-device-adapt/>), ainda em rascunho enquanto escrevo o livro.

A principal diferença é que essa spec joga a responsabilidade para o cara certo, o CSS, em vez de uma tag HTML. É a nova regra `@viewport` que você pode usar assim:

```
@viewport {  
    width: device-width;  
}
```

Lembra bastante a meta tag e tem parâmetros parecidos (consulte a spec pra mais detalhes). Uma vantagem bem interessante é poder usar várias configurações de viewport ao mesmo tempo, definidas com media queries. A sintaxe é simples mas dá um nó na cabeça misturar tudo isso. Reflita sobre o que o código a seguir vai fazer:

```
@media (max-width: 400px) {  
  @viewport {  
    width: 320px;  
  }  
}
```

Para telas até 400px de largura, o viewport é fixado em 320px. Isso quer dizer que o site é renderizado a 320 CSS pixels e depois redimensionado pra encaixar na resolução real da tela (que tem menos de 400px). É um jeito de fazer o layout encaixar na tela sem fazer um layout fluído de verdade — o browser redimensiona sozinho nosso layout fixo em 320px pra encaixar na tela. Daria então pra fazer 3 designs (smartphone, tablet, desktop) fixos, com viewports diferentes e deixar o browser redimensionar baseado no tamanho da tela.

Tudo isso será futuro, pois o suporte nos navegadores é bem baixo enquanto escrevo o livro. O IE10 é o primeiro a usar essa regra (ainda com prefixo) num modo novo do Windows 8 chamado *Snap Mode* (<http://timkadlec.com/2012/10/ie10-snap-mode-and-responsive-design/>).

CAPÍTULO 11

A saga dos 3 pixels e as telas de alta resolução e retina

No tópico anterior sobre viewport (10) falamos da diferença entre *pixels físicos* e CSS *pixels*. Pois bem, o mundo era desse jeito até começarem a surgir os primeiros Androids com telas de alta resolução e iPhones retina. Temos 3 pixels diferentes agora.

Os iPhones antigos vinham com 320px de largura. A partir do iPhone 4, o primeiro com tela retina, eles passaram a vir com o dobro de resolução. Ou seja, 640px de largura. Mas o tamanho físico do aparelho é o mesmo. O que acontece então?

Como ficam nossas páginas mobile então que assumiam uma resolução bem menor? Com resolução tão alta quanto um Desktop, os celulares mais modernos vão renderizar as páginas bem pequenas, como um site Desktop? Felizmente, nossas páginas continuam funcionando porque **esses dispositivos de alta resolução continuam reportando um device-width de 320px**, pra manter a compatibilidade.

A ideia de reportar um device-width diferente do tamanho de pixels físicos surgiu no Android e depois foi copiada pelo iOS e outras plataformas. Dessa forma,

é possível evoluir a resolução da tela com densidades de pixels maiores sem afetar a forma como o usuário usa nosso Site mobile, que continua otimizado para telas pequenas.

Devemos encarar o `device-width` como o tamanho que dará melhor usabilidade para o usuário.

Mas tudo isso causa muita confusão. Aliás, já li muita matéria em sites conceituados como Gizmodo ou Smashing Magazine fazendo confusões gigantescas com esses conceitos. Por isso, vamos entender melhor.

Resoluções e DPIs

Duas medidas são importantes numa tela: o **tamanho físico** da tela em centímetros — ou polegadas, como é mais comum — e sua **resolução em pixels**. Da divisão desses dois números, temos o **DPI** — **dots per inch** — que diz quantos pixels existem por polegada de tela.

Conseguimos saber a resolução em pixels da tela em JavaScript com as propriedades `screen.width` e `screen.height`. Mas, com JavaScript, **não é possível saber o tamanho físico da tela e nem seu DPI**.

O curioso é que o CSS tem algumas unidades para tamanho físico — você pode criar um `div` com `width:1cm` ou `1in`. Mas isso não faz o que realmente queremos, que seria um elemento de 1 centímetro ou 1 polegada *física*. Os navegadores em geral concordaram em um **dpi base de 96** e todas as contas são relativas a esse número. Então, se fizer um `div` de 1 polegada, na verdade você leva um de 96px, não interessando o tamanho físico real da tela ou seu dpi verdadeiro.

Pixel ratio, CSS pixels e telas retina

Tela retina é o termo que a Apple usa pra falar de telas de alta resolução. São telas com tantos pixels que eles acabam sendo pequenos fisicamente, tornando difícil o olho identificar um individualmente. Com isso, o conteúdo mostrado ficaria muito pequeno na tela, então a grande novidade é que **um pixel de conteúdo é renderizado com mais de um pixel físico**. *Ähn?*

A ideia é simples: se você desenhar uma linha de 1px na tela (com CSS ou mesmo uma app nativa), ela será renderizada como uma linha de 2px físicos. Isso melhora, e muito, a definição dos elementos na tela. Existe, então, uma *razão de multiplicação* entre pixels lógicos e pixels físicos. Essa razão é o **device pixel ratio**, acessível em JS pela propriedade `devicePixelRatio` e nas media queries CSS de alta resolução.

No mundo Apple, as telas retina têm, hoje, sempre *device pixel ratio* igual a 2, o que facilita as contas. No Android, essas mesmas telas são chamadas de **xhdpi** — como num Galaxy S3 ou num Nexus 4. Mas, no Android, temos mais possibilidades:

- xhdpi (extra high dpi) - **pixel ratio 2** - ex. Galaxy S3, Galaxy Note 2, Nexus 4.
- hdpi (high dpi) - **pixel ratio 1.5** - ex. Galaxy S2, Motorola Atrix, Nexus One.
- tvdpi - **pixel ratio 1.33** - Nexus 7.
- mdpi (medium dpi) - **pixel ratio 1** - telas normais, comuns em smartphones simples.
- ldpi (low dpi) - **pixel ratio 0.75** - aparelhos low-end como o Galaxy 5.
- dpi altíssimo (ainda sem nome) - **pixel ratio 3** - aparelhos Full HD com tela 1920x1080 como o HTC One e o Galaxy S4.

(Curioso notar o ldpi com pixel ratio menor que 1. Nesse caso, o conteúdo é desenhado com menos pixels na tela. Uma linha de 4px no CSS vai ser desenhada com 3 pixels físicos. Esse downscale deixa a renderização visualmente pior, claro.)

DIP, o terceiro pixel

No tópico anterior (10), falamos de **CSS pixel**, que representa as medidas dos elementos da página, e de **pixels físicos**, que são os pequenos pontos da tela.

Quando a tela tem *device pixel ratio* diferente de 1, temos um terceiro conceito, chamado de **DIP** — *device independent pixel*, conhecido também como *UI pixel*. Esse é o pixel lógico.

Falamos que a tela do iPhone mede 320px em DIP, apesar de medir 640px físicos. Isso porque o *device pixel ratio* do iPhone retina é 2.

Quando usamos o viewport com `width=device-width`, 1 CSS pixel mede 1 DIP. Mas lembre que podemos abrir um site Desktop no nosso iPhone e aí, nesse caso, a página mede 980 CSS pixels, 320 DIPs e 640 pixels físicos. É essencial que, nesse ponto, essa última frase faça sentido na sua cabeça. Três pixels diferentes, pra confundir todo mundo!

E, claro, a relação entre DIP e pixels físicos é: **número de DIPs × devicePixelRatio = número de device pixels**. Já a relação entre CSS pixel e DIP tem a ver com o viewport que estamos usando. Como, na maioria dos sites otimizados pra mobile, vamos usar `device-width`, então: **um CSS pixel = um DIP**.

O que importa é o tamanho em DIP

Todos os iPhones mostram 320px de conteúdo, não importando se é retina (com 640px físicos) ou não. Precisa ser assim, pois os pixels físicos são muito pequenos na tela retina. Seria péssimo pro usuário mostrar 640px de conteúdo, tudo ficaria muito pequeno.

No fim, pra usabilidade, **conta mais o tamanho da tela em DIP que o tamanho físico.**

Não quer dizer claro, que ter uma tela com alta resolução e com alto *pixel ratio* seja ruim. Embora o conteúdo fique igual de tamanho nos dois tipos de iPhone, ele fica **mais bem definido** no retina, claro. Textos e gráficos ficam ótimos. Conteúdos grandes e com *zoom out* ficam mais definidos — como uma foto grande ou um vídeo HD ou mesmo um site Desktop visto no celular.

Devíamos falar de ‘DPI de conteúdo’

A confusão maior nisso é que as especificações dos aparelhos e matérias nos sites de tecnologia costumam falar apenas da resolução física do aparelho e não de viewports e pixel ratios. Claro, esses conceitos confundem muita gente, principalmente o usuário final. Mas a contradição é que **o viewport é justamente o número mais útil pro usuário, muito mais que saber a resolução física.**

Num artigo que li recentemente no Gizmodo, havia uma comparação entre o iPad Mini e o Nexus 7. O argumento era que a tela de 1024x768 do iPad Mini era pior que a de um Nexus 7 de 1280x800 pois tinha *menor área útil disponível para o usuário*. Errado! Embora com resolução física mais alta, o Nexus 7 tem um *device pixel ratio* 1.33x e viewport menor (966x603px contra 1024x768px do iPad Mini). Um usuário que queira ver mais conteúdo na tela, vai preferir o iPad Mini.

Quando lemos comparativos sobre aparelhos, é comum ver as pessoas citando DPIs físicos. Por exemplo:

- iPhone não-retina tem **163 dpi** com largura de **320px**
- iPhone retina tem **326 dpi** com largura de **640px**
- Galaxy S tem **233 dpi** com largura de **480px**
- Galaxy Y tem só **133 dpi** com largura de **240px**

Só vendo esses poucos exemplos, a ideia é que temos telas com resoluções bem diferentes e DPIs diversos.

Mas isso não é verdade se levarmos em conta a área real pro conteúdo em DIP. Por isso, acho que deveria existir uma métrica de **dpi de conteúdo**, que mostra a densidade da tela com base na quantidade de conteúdo que ela mostra, em *device independent pixels*.

Olhando novamente pra lista de aparelhos vemos que todos têm a mesma largura de viewport e a variação na densidade de DIP é bem menor:

- iPhone não-retina tem **163 dpi de conteúdo** com viewport de **320 DIP** de largura
- iPhone retina tem **163 dpi de conteúdo** com viewport de **320 DIP** de largura
- Galaxy S tem **155 dpi de conteúdo** com viewport de **320 DIP** de largura
- Galaxy Y tem **177 dpi de conteúdo** com viewport de **320 DIP** de largura

Pro usuário final, essa medida de *dpi de conteúdo* é mais importante que o *dpi físico*.

Porque um iPad mini tem o mesmo viewport que um iPad normal?

O lançamento do iPad mini trouxe muita discussão sobre sua usabilidade. O Jakob Nielsen, papa da usabilidade, declarou que a tela é pequena demais e com problemas de UX (<http://www.useit.com/alertbox/specifications-vs-ux.html>). Outros reviews chegaram na mesma conclusão, e até nomes como Luke Wroblewski e Brad Frost entraram nessa discussão.

O problema do iPad Mini é que ele tem a mesma resolução de 1024x768 do iPad normal mas em uma tela 27% menor fisicamente. Pior, seu device pixel ratio é 1, o que faz com que o viewport final seja também de 1024x768. Na prática, todas as coisas do iPad abrem 27% menores em tamanho, tornando os botões mais difíceis de apertar, os textos menores pra ler e etc.

Muita gente se pergunta em como a Apple chegou no tamanho de 7.9 polegadas do iPad Mini. Esse número não é aleatório. Com 7.9, **o iPad mini tem o mesmo dpi de conteúdo dos iPhones**. Ou seja, vai renderizar tudo do mesmo tamanho físico que um iPhone faz. Veja as contas:

- iPhone não-retina (320x480) tem **162.98 dpi de conteúdo** numa tela de 3.5” com pixel ratio 1

- iPhone retina (640x960) tem **162.98 dpi de conteúdo** numa tela de 3.5” com pixel ratio 2
- iPad mini (1024x768) tem **162.02 dpi de conteúdo** numa tela de 7.9” com pixel ratio 1
- iPad 4 retina (2048x1536) tem **132 dpi de conteúdo** numa tela de 9.7” com pixel ratio 2

Olhando essa lista, percebemos que o estranho é o iPad normal, com um dpi bem mais baixo. As coisas ficam maiores fisicamente no iPad normal, o que é bom já que um tablet grande costuma ser usado a uma distância maior do usuário (apoiado na mesa, no colo etc).

Na prática, o que a Apple está dizendo é que o **iPad mini foi feito para ser usado na mesma distância que você usa o iPhone**, e não como você usa o iPad normal.

Quantos pixels minha tela deveria ter?

A questão aqui é de **usabilidade**. Pixels pequenos demais tornam a leitura do usuário mais difícil, que é grande crítica dos analistas ao iPad mini. Você precisa usar o aparelho mais perto do rosto, o que pode não ser muito natural.

O W3C tem uma medida oficial pra determinar o tamanho de um pixel de conteúdo na tela, o que eles chama de **pixel de referência** (<http://www.w3.org/TR/css3-values/#reference-pixel>). Ele é medido em ângulos, claro. Um pixel num celular que usamos perto do olho tem comprimento menor que um pixel numa televisão que vemos a metros de distância. Para analisar se um pixel está do “tamanho certo”, é preciso saber a **distância do usuário em relação à tela**.

O tamanho do pixel de referência do W3C é de *0.0213 graus*. Deixando a matemática um pouco de lado, isso quer dizer que:

- Um **iPhone** foi projetado para ser visto a uma distância de **41.9cm** do olho;
- O **iPad normal** é melhor visto a uma distância de **51.7cm**;
- Já o **iPad Mini** é pra ser usado como um iPhone, a **42.1cm** de distância.

(se quiser saber melhor sobre a matemática por trás disso tudo, recomendo: <http://www.kybervision.com/Blog/files/AppleRetinaDisplay.html>)

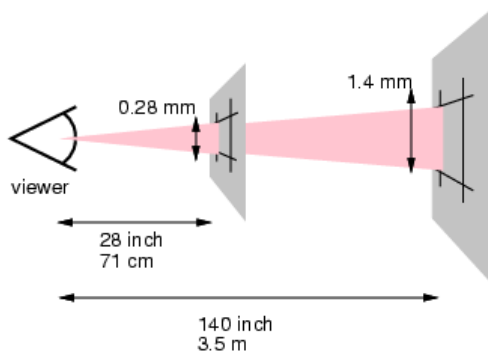


Figura 11.1: Medida do pixel de referência do W3C (imagem oficial W3C)

Celulares Android são os que trazem a maior variação nos DPIs. Muitos trazem telas grandes fisicamente mas com *dpis de conteúdo* dos mais variados.

- Um **Galaxy S3** tem 360x640 de viewport numa tela de 4.8". É melhor usado a **44.6cm** de distância;
- Já um **Galaxy Note 2** tem uma tela de 5.5" com a mesma resolução física e mesmo viewport — 360x640 — do S3. Ou seja, é um aparelho pensado para ser usado a uma distância maior, de **51.17cm**, como um iPad, apoiado na mesa por exemplo.

E, claro, toda essa usabilidade depende de usuário para usuário. **Depende de como você pretende usar aparelho e da qualidade da sua visão.** O tamanho do *pixel de referência* é calculado pensando na resolução média do olho humano. E existe toda uma matemática interessantíssima sobre a ótica do olho humano pra se calcular isso (<http://blogs.discovermagazine.com/badastronomy/2010/06/10/resolving-the-iphone-resolution/>).

Por exemplo: a 30cm de distância, a percepção máxima do nosso olho é, em média, de 286dpi. Qualquer coisa acima disso, e a maioria das pessoas não conseguirá distinguir os pixels individualmente. Por isso, a tela de 326dpi do iPhone retina é bastante definida pra gente. Mas o olho humano consegue atingir resolução máxima de 477dpi em quem tem visão perfeita; nesse caso, um celular FullHD de 469dpi como o **HTC One** pode ser melhor.

Conclusão

Como desenvolvedor, para trabalhar com telas retina, você estará sempre prestando atenção ao `devicePixelRatio` e ao tamanho da tela em DIP. O tamanho físico da tela não importa muito. Claro, tudo isso vai afetar como você cria suas media queries e imagens. Vamos ver media queries de alta resolução no tópico [15](#) e como preparar imagens para telas retina no tópico [19](#).

Como usuário, você deve escolher seu aparelho baseado no tamanho da tela em DIP e a forma que pretende usá-lo. Prefira aparelhos com *device pixel ratio* alto pois tudo fica mais definido. Mas, principalmente, escolha um aparelho que tenha um tamanho em DIP que seja confortável ao seu olho e à distância que você quer usá-lo.

E, se precisar, você também pode dar zoom enquanto navega pra ver as coisas melhores. É o assunto do próximo tópico [12](#).

CAPÍTULO 12

Não remova o zoom dos seus usuários

Vimos no tópico sobre **viewport** (10) a importância da meta tag para adaptar a página corretamente nos dispositivos móveis. Analisamos os parâmetros `width` e `initial-scale`, mas há outros. Os demais parâmetros controlam o gesto de redimensionar a página (*page scale*). Com eles, você consegue bloquear o zoom do usuário de algumas formas:

```
<meta name="viewport" content="width=device-width, user-scalable=no">
```

```
<meta name="viewport"  
      content="width=device-width, minimum-scale=1, maximum-scale=1">
```

Mas, na esmagadora maioria dos casos, você não deveria fazer isso. Exceções talvez sejam webapps com uma ideia de canvas fullscreen onde os gestos são tratados pela aplicação (mapas e jogos talvez). Em sites web normais? **Jamais.**

As telas pequenas dos smartphones ensinaram algo simples para os usuários: se algo estiver pequeno, apenas pince os dedos (*pinch*) e **dê zoom!** É um gesto básico de dispositivos touch e conhecido por todo mundo. Mas, mesmo assim, muitos sites bloqueiam o zoom nas páginas. **Não faça isso.**

Há um mito que circula por aí de que limitar o zoom faz com que nossa página fique mais parecida com uma *App*. Primeiro: site não é App, então não tente parecer uma. Segundo: se algumas Apps têm essa *limitação* de não deixar dar zoom, por que copiar essa *deficiência* pra sua página?

Controle na mão do usuário

Desabilitar o zoom das páginas é tão irritante, mas tão irritante, que os browsers mobile modernos estão deixando esse controle na mão do usuário! O browser do **Android 4** e o **Chrome Mobile**, por exemplo, têm essa opção nas configurações:

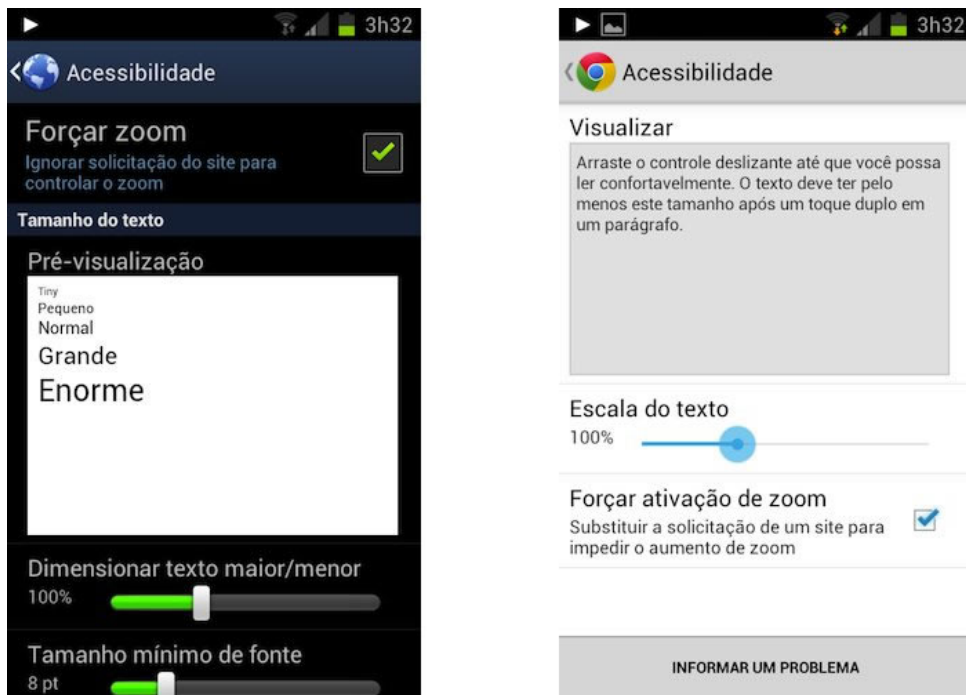


Figura 12.1: Configurações de zoom do Android 4 e do Chrome Mobile

O Mobile Safari do iOS, infelizmente, ainda não tem uma opção dessas. Também não consegui achar no Opera e no Firefox uma opção semelhante. Uma gambiarra útil pra usuários dessas plataformas é usar um **bookmarklet que reescreva a tag viewport** dos sites pra habilitar o zoom sempre (<https://gist.github.com/sergiolopes/5231024>).

O famoso bug do zoom no iOS

Eu acho que o grande culpado da proliferação de páginas com zoom desabilitado é um famoso bug no iOS até a versão 5.x que faz com que a página dê um zoom quando você gira o aparelho em modo paisagem. É bem irritante. Você está vendo a página em modo retrato e, quando gira o aparelho, a Safari dá um zoom in e você não consegue ver a página toda — e precisa fazer um gesto de zoom out.

Se desabilitarmos o zoom, o bug não acontece. Mas é um jeito covarde de resolver o problema. O **iOS 6** resolve esse bug do Mobile Safari, então isso é coisa do passado.

Mas mesmo que o bug do iOS seja um problema pra você e seus usuários com iOS velho, pense em alguma opção:

- **Não faça nada.** Sim, uma opção é deixar o bug acontecer. Lembre que um usuário de iOS está *acostumado* a isso, afinal todos os sites do mundo são afetados! E a maioria dos usuários com iOS 6+ está ok.
- Se incomodar muito e você quiser tirar o zoom do usuário por causa do bug, pelo menos faça isso **apenas no iOS** e não limite todos os outros dispositivos do mundo que funcionam direito. Lembre que, principalmente no Brasil, o Android é muito mais usado que o iOS.
- Há **hacks em JavaScript** que solucionam o problema em 99% dos cenários. Veja um comparativo de hacks que resolvem o bug: <https://github.com/sergiolopes/ios-zoom-bug-fix#other-solutions>

Conclusão

De modo geral, desabilitar a possibilidade do usuário dimensionar a página a seu gosto é péssima usabilidade. Nem todo usuário tem visão perfeita e o gesto de *page scale* é uma das ferramentas essenciais de acessibilidade. Vamos discutir mais sobre acessibilidade relacionada ao outro tipo de zoom, o *page zoom*, em outro tópico (16).

CAPÍTULO 13

Use sempre media queries baseadas no conteúdo da sua página

Ao escrever medias queries, você precisa escolher algum *valor* pra colocar lá. É o que chamamos dos **breakpoints**, os pontos onde seu layout vai ser ajustado por causa de uma resolução diferente. E **escrever bons breakpoints** é essencial para um design responsivo de qualidade.

O que mais aparece de pergunta de quem está começando com design responsivo é: *quais os valores padrões de se colocar nas media queries?* E logo surge uma lista parecida com essa:

```
@media only screen and (min-width: 480px) { ... }  
@media only screen and (min-width: 600px) { ... }  
@media only screen and (min-width: 768px) { ... }  
@media only screen and (min-width: 992px) { ... }  
@media only screen and (min-width: 1382px) { ... }
```

Péssima prática! Essa lista eu copiei do famoso projeto *320andup* mas você acha

outras parecidas, com esses valores supostamente “comuns”. *480px* é por causa de um iPhone em landscape, *600px* é comum em tablets de 7”, *768px* pega um iPad em portrait, *992px* é um Desktop de 1024 menos as barras de rolagem e *1382px* pra Desktops grandes.

O pessoal chama essa prática de **device-driven breakpoints**, pois são valores gerados a partir de tamanhos de dispositivos.

Não use device-driven breakpoints

Primeiro problema: essa lista pensa em meia dúzia de tipos de dispositivos. Mas e os *360px* de um Galaxy S4? Ou os *533px* do Galaxy SII em landscape? Ou o iPhone 5 com *568px* em landscape? Qualquer lista de media queries padrão é **muito pobre** quando pensamos na situação atual — e, ainda mais, futura — dos diferentes dispositivos.

Segundo problema: quem disse que *seu design* realmente precisa de todos esses breakpoints? Ou pior, que ele não vai ficar ruim entre as paradas de *480px* e *600px*? Usar esses valores de media queries **não garante que seu design funcionará em todos os dispositivos**, mas apenas nos dispositivos “padrões”, seja lá o que for isso.

Use content-driven breakpoints

As media queries a seguir eu tirei do CSS do meu blog pessoal:

```
@media (min-width: 592px) { ... }
@media (min-width: 656px) { ... }
@media (min-width: 752px) { ... }
@media (min-width: 1088px) { ... }
@media (min-width: 1312px) { ... }
```

Da última vez que chequei, não consegui encontrar nenhum dispositivo com *656px* ou *752px* de largura. De onde saíram então esses valores? Do design do meu blog.

A ideia é simples: fiz meu design responsivo com medidas flexíveis e pra pequenos dispositivos primeiro — *mobile first*. Aí fui testando as diversas resoluções pra ver onde o design não ficava tão bom e coloquei um breakpoint lá.

É o que o pessoal chama de **content-driven breakpoints**. Ou seja, achar suas media queries a partir do seu conteúdo e do seu design. Fica bem mais fácil garantir que sua página **funcione em todos os dispositivos**.

Como achar meus breakpoints?

- Abra sua **página original** no navegador;
- Vá **redimensionando a janela devagar** até o design parecer ruim — se fez *mobile-first*, abra pequeno e vá aumentando a janela; senão, abra grande e vá diminuindo a janela;
- Quando achar um ponto em que o design quebra, copie o **tamanho da janela** e crie uma media query com esse valor lá no seu CSS;
- Recarregue a página, veja se as mudanças melhoraram o design, e continue redimensionando pra achar o **próximo breakpoint**.

Algumas ferramentas que podem te ajudar nisso:

- O *responsive mode* do Firefox (https://developer.mozilla.org/docs/Tools/Responsive_Design_View);
- **FitWeird** (<http://davatron5000.github.com/fitWeird/>), um excelente bookmarklet que te dá o tamanho da tela, inclusive em em;
- Meu **responsive play** (<http://sergiolopes.org/responsive-video-play/>).

Tweakpoints

Esses pontos em que nosso design quebra e colocamos uma media query são comumente chamados de **breakpoints** no mundo da web mobile. E, se você observar tanto o exemplo do meu blog quanto o das resoluções de aparelhos comuns, vai notar que existem 5 *media queries*. Essa coincidência não foi de propósito, mas acaba mostrando uma outra questão: como é comum ver projetos com poucas media queries, meia dúzia de breakpoints em geral.

Não precisava ser assim. Aliás, o próprio nome de *breakpoint* dá a impressão de que devemos criar uma media query apenas quando o layout *quebra*, quando ele muda radicalmente. É comum suportar uma versão do nosso design numa coluna só, talvez uma outra versão com 2 colunas e outra com 3 colunas. Essas grandes quebras exigem bastante código CSS pra alterar o layout significativamente.

Mas, muitas vezes, a gente só precisa de um pequeno ajuste no design. Aumentar levemente a fonte de um título a partir de certo ponto. Ou trocar a fonte

no menu para bold em telas maiores. Situações como essa podem acontecer em qualquer ponto do nosso design, não necessariamente nos *breakpoints* maiores. E não há nada de errado em criar media queries mais simples, com pequenos ajustes, em diversos pontos intermediários. O Jeremy Keith chama isso de **tweakpoints** (<http://adactio.com/journal/6044/>), e é um bom nome.

Da última vez que contei no código fonte do site da Caelum, por exemplo, haviam 33 *valores diferentes em media queries*. Todas baseadas no conteúdo mas várias apenas com pequenos ajustes e uma única linha de CSS dentro, *tweakpoints*.

CAPÍTULO 14

Media queries mobile first ou desktop first?

Imagine o seguinte cenário: você tem um texto e uma foto que flutua à direita, com o texto passando ao redor. Nada de mais, algo bastante comum. Mas em telas pequenas, o design fica *estranho*. A imagem ocupa quase toda a largura da tela e o texto flui bizarramente ao redor.

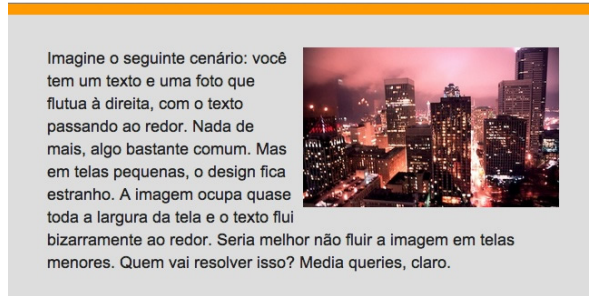


Figura 14.1: Texto flutuando ao redor da imagem numa tela grande.

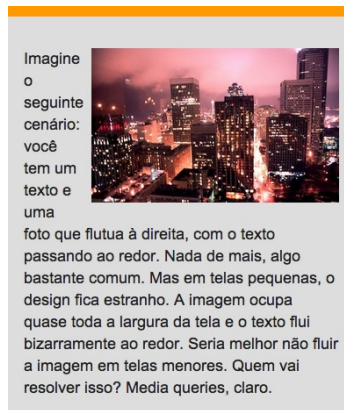


Figura 14.2: Texto flutuando bizarramente se a tela for menor.

É bem fácil melhorar isso. Seria melhor não fluir a imagem em telas menores. Quem vai resolver isso? *Media queries*, claro. Se a imagem tiver uma classe `foto`, por exemplo, basta fazer `float:right` em uma situação e `float:none` em outra.

Mas, se pararmos pra pensar bem, há duas formas de resolver essa questão. A primeira é fazer a imagem flutuar e, depois, com uma *media query* que selecione telas pequenas, tirar o `float`:

```
/* a foto flutua a direita sempre */
.foto {
  float: right;
}
```

```
/* mas em telas com no máximo 550px, melhor não flutuar */
@media (max-width: 550px) {
  .foto {
    float: none;
  }
}
```

A outra forma é inverter as media queries, e trabalhar com min-width:

```
/* a foto não flutua, então não preciso
   de CSS já que float:none é o padrão */
/* ... */

/* em telas maiores, quero que flutue */
@media (min-width: 551px) {
  .foto {
    float: right;
  }
}
```

Os dois códigos fazem exatamente a mesma coisa. Mas você deve ter reparado que o segundo é mais simples e curto, já que evita sobrescrever uma propriedade no CSS.

Aliás, **sobrescrever regras no CSS é algo perigoso, evite isso**. Você escreve a mesma propriedade duas vezes, repetindo código. Ou, pior ainda, sobrescreve uma propriedade sem aproveitar o padrão do browser — como um `float:none` ou um `width:auto`. É sinal de que você aplicou o CSS anterior em mais elementos que gostaria e, agora, precisa sobrescrever. Indica que você escolheu errado seu seletor e o ideal seria invertê-lo pra aplicar o valor mais específico só nos elementos que realmente precisam disso.

Essa dica de evitar sobrescrever regras vale pra todo CSS. Mas eu vejo muita gente se perdendo nelas justo quando fazem as media queries. É muito fácil cair no primeiro código que mostrei antes, justificando que a sobrescrita é porque mudamos o layout em telas pequenas.

O primeiro código é um exemplo de código CSS **desktop-first** enquanto o segundo usa uma abordagem **mobile-first**.

Media queries mobile first

Já falamos de *mobile first* antes no tópico 6. Mas a ideia aqui é pensar em como isso afeta a construção das nossas media queries no código CSS.

Usando *media queries mobile first* você cria seu código inicial focado no design pra menores telas, de smartphones. O design dessa versão deve ser um pouco mais simples que das versões maiores, então seu código deve ser mais conciso. Depois, vá criando media queries para ir transformando e incrementando seu design para telas maiores. Nessa técnica, você usa várias media queries *min-width*, sempre pensando nas telas maiores.

Essa metodologia simplifica bastante seu código CSS. Na abordagem contrária, *desktop-first*, a gente acaba escrevendo um monte de media queries com *max-width* que sobrescrevem as regras anteriores. Isso porque o design base é mais complexo, pensado no Desktop, e deve ser simplificado para telas menores.

Não suportar media queries é a primeira media query

Essa excelente frase é bastante encontrada na comunidade mobile. Resume o melhor do espírito *mobile-first* e *progressive enhancement*.

Se o seu dispositivo não suporta media queries, isso é um forte indicativo pra você das capacidades mais limitadas do mesmo. Nesse cenário, se você criou seu CSS *mobile-first*, você garante que a experiência padrão nesse aparelho será a mobile mais simples de todas. Esse é seu CSS padrão, fora das media queries. Se tivéssemos feito o código *desktop-first*, o design aplicado na falta de suporte a media queries seria o desktop, não uma boa experiência.

Nesse sentido, você pode encarar a existência de suporte a media queries como uma melhoria, *progressive enhancement*. A experiência padrão é aceitável, mas ter media queries fará o design ser um pouco mais bem adaptado dependendo do aparelho.

Browsers antigos e polyfills

Mas, claro, às vezes você quer aplicar as media queries em navegadores importantes mas que ainda não as suportam. O exemplo clássico são as versões do Internet Explorer antes da 9 e versões bem antigas do Firefox. O uso desses navegadores hoje é bem baixo, principalmente no Brasil. Mas, dependendo do seu projeto, pode ser importante ter um suporte a essas versões. Pensando nisso, é possível usar algumas estratégias.

A primeira é **não fazer nada**. Se você fez seu design mobile first e flexível, a experiência de navegação num navegador sem media queries é aceitável. Pode não ser o design padrão que você pensou para Desktop, mas é aceitável e usável. Lembre, encare a existência de media queries como uma *melhoria*, não como uma *exigência*. Não fazer nada é uma boa solução se você tem poucos visitantes desses navegadores velhos e não quer gastar muito tempo e dinheiro implementando uma solução pra tão pouca gente.

Outra abordagem é ter um segundo CSS que aplica todas as regras de CSS sem media queries e carregá-los somente nos IEs velhos do Desktop com comentários condicionais. O ideal seria gerar esse segundo CSS automaticamente usando algum pré-processador ou algum script seu. E, para os navegadores normais e o IE mobile, você inclui o CSS normal escrito de forma *mobile-first*:

```
<!--[if (gt IE 8) | (IEMobile)]><!-->
    <link rel="stylesheet" href="estilo-normal.css">
<!--<![endif]-->

<!--[if (lt IE 9) & (!IEMobile)]>
    <link rel="stylesheet" href="estilo-ie.css">
<![endif]-->
```

A vantagem dessa solução é sua simplicidade e performance, uma vez que cada navegador recebe o CSS adequado às suas capacidades. O ponto negativo é que só serve para IEs velhos, e não para outros navegadores velhos — o Firefox 3 não suporta media queries, por exemplo.

A terceira abordagem para suportar media queries nesses navegadores velhos é fazê-los entender as media queries. Usar um polyfill em JavaScript que lê seu código CSS e interpreta as regras das media queries, aplicando as que forem apropriadas. As soluções mais famosas são o **Respond.js** (<https://github.com/scottjehl/Respond>) e o **CSS3-MediaQueries.js** (<https://code.google.com/p/css3-mediaqueries-js/>). Esse último é o mais completo, suportando diversas medias queries em todo tipo de navegador velho.

Já o *Respond.js* é mais simples e só suporta `min-width` e `max-width`, o que também faz ele ser mais leve e rápido, ideal para lidar com navegadores velhos. Como usamos só essas media queries na maior parte dos casos, pode ser mais interessante. Para usá-lo, apenas referencie o script na página.

CAPÍTULO 15

As media queries para resoluções diferentes e retina

Muitas vezes, você vai querer carregar imagens e estilos diferentes em telas de resolução diferentes ou retina. Para isso, você precisa de **media queries de resolução**.

Nesse tópico, discuto a forma correta de se atacar telas de alta resolução com CSS, as diferenças dos navegadores, que valores usar e algumas boas práticas. O resumo do tópico é que você deve usar uma **media query parecida com essa**:

```
@media (-webkit-min-device-pixel-ratio: 1.5),  
       (min-resolution: 144dpi) {  
  
}
```

Sim, eu sei que você provavelmente já viu ou usou uma versão bem maior dessa media query, cheia de prefixos de navegadores e valores estranhos. Como essa:

```
@media (min--moz-device-pixel-ratio: 1.5),  
       (-o-min-device-pixel-ratio: 3/2),
```

```
(-webkit-min-device-pixel-ratio: 1.5),  
(min-device-pixel-ratio: 1.5),  
(min-resolution: 144dpi),  
(min-resolution: 1.5dppx) {  
  
}
```

Pois bem, não use mais essa versão longa e complicada. O **correto para os browsers de hoje e do futuro** é a primeira que mostrei. Vejamos o que ela faz.

Por que essa media query pra telas retina?

Muito tempo atrás, quando surgiram as primeiras telas com *pixel ratio* diferente de 1, o pessoal do WebKit criou uma media query pra isso, a famosa `-webkit-device-pixel-ratio` (e suas versões com `min` e `max`). Alguns browsers chegaram a experimentar isso, surgindo até a famosa aberração da Mozilla com `min-moz-device-pixel-ratio` (não, não há um erro de digitação nessa sintaxe bizarra com dois traços).

Mas aí veio a especificação oficial e revisaram essa media query, criando uma equivalente com um nome mais simples: `resolution`. Você pode escrever `min-resolution` e `max-resolution` e ela aceita várias medidas, sendo **dpi** e **dppx** as mais importantes.

Cuidado, porém, com os valores das medidas. Você já leu o tópico sobre resoluções e DPI (11) então lembra que os **browsers fixam a resolução lógica em 96dpi**, independente da resolução física. Então, quando escrevi **144dpi** na media query lá em cima, o que estava dizendo é que queria 1.5x o DPI padrão de 96, pegando então telas com *device pixel ratio* 1.5, como vários aparelhos Android.

A medida **dppx** é nova no CSS e significa o mesmo que o *device pixel ratio*. Então, poderíamos escrever **1.5dppx** e seria equivalente a **144dpi**.

Qual valor usar? Realmente preciso de imagens diferentes pra cada resolução?

Os valores que encontramos no mercado hoje são:

- **0.75dppx** (= 72dpi) - Android low-end, tipo Galaxy 5;
- **1dppx** (= 96dpi) - Notebooks, desktops e vários celulares e tablets;
- **1.33dppx** (= 127dpi) - Nexus 7;

- **1.5dppx** (= 144dpi) - Vários Androids, como Atrix ou S2;
- **2dppx** (= 192dpi) - Telas retina da Apple, celulares e tablets mais modernos como S3;
- **3dppx** (= 288dpi) - Celulares ultra modernos, como HTC One e Samsung Galaxy S4.

Você pode escrever uma media query `resolution` para cada um desses valores, servindo imagens adaptadas para cada resolução. Pode até ser uma ideia se você for gerar tudo isso automaticamente (imagens e media queries), mas, na mão, fica inviável.

O que vejo muito comum de se fazer é **servir uma imagem em alta resolução para qualquer tela acima de 1dppx**. Você pode fazer algo como o pessoal do Word-Press que escolheu **120dpi** (= 1.25dppx) como valor inicial pra servir imagens retina.

Mas veja se realmente vale a pena servir imagens diferentes só por causa do DPI. Isso vai ficar bastante complicado nos próximos anos. **Para suportar bem todo tipo de tela, prefira usar texto, fontes, SVG e CSS**. Para fotos, há outras técnicas. Vamos discutir a melhor forma de atacar imagens responsivas no tópico [19](#).

E os browsers?

Todos os browsers modernos **suportam a media query resolution**, exceto os baseados no WebKit. Existem ainda alguns outros problemas, principalmente em dispositivos móveis. Alguns testes feitos no início de 2013:

- **Firefox e Opera** no Desktop já suportam `resolution`.
- No **WebKit**, já existe suporte desde Outubro/2012 mas isso ainda não chegou no browsers finais. Por isso, usamos ainda a antiga `-webkit-device-pixel-ratio`, pensando no Chrome, Safari, Android e iOS.
- **Opera Mobile** suporta `resolution` nativamente no Android. Único defeito é que ele reporta pixel ratio 1 no meu Galaxy 5 de pixel ratio 0.75. Mas nos celulares com 1.5 e 2, tudo funciona perfeito.
- **Firefox no Android** só aceita a velha `-moz-device-pixel-ratio`, mas ele devolve sempre pixel ratio 1 e não suporta telas retina ainda. Isso é um bug e,

quando for consertado, fará a `resolution` funcionar, como no Firefox 18 pra Mac que já suporta o MacBook Retina.

- **Opera Mini** reporta sempre pixel ratio **1**, tanto no Android quando no iOS, mesmo em telas retina. O problema é que ele suporta a media query `resolution` cheia de bugs: se usar `dppx` ele aceita qualquer valor, e o `min-resolution` acha que o pixel ratio é 2. Conclusão: o suporte nele ainda é bem bugado, mas acrescentar `-o-device-pixel-ratio` não vai melhorar isso.
- **Internet Explorer** suporta `resolution` desde a versão 9 e nunca suportou `device-pixel-ratio`.
- Outros browsers que testei: *Dolphin Android Beta* e *UC Browser*. Em ambos, funciona `-webkit-device-pixel-ratio` mas eles sempre reportam **1**, mesmo em telas retina.

A conclusão de tudo isso é que precisamos ainda da `-webkit-device-pixel-ratio` mas já podemos usar a `resolution` para os outros browsers, sem prefixos. O ideal também é usar a unidade **dpi** e **não dppx**, já que o suporte nos browsers ainda é precário.

Só lembre que essa medida **dpi** usada na media query é relativa a polegadas de CSS, que são padronizados em 96px. Isso não é o DPI físico do aparelho. Já vi muita gente se confundir com isso. Um iPhone Retina, por exemplo, vai reportar sempre **192dpi** (2x o dpi base de 96) e nunca seu dpi real, dos pixels físicos (que seria 330).

A media query final

Ta aí então a media query final pra suportarmos telas de alta resolução hoje e no futuro, sem problemas:

```
@media (-webkit-min-device-pixel-ratio: 1.25),
  (min-resolution: 120dpi) {
```



```
}
```

CAPÍTULO 16

Media queries também ajudam na acessibilidade

Sempre que alguém fala em medias queries, a primeira coisa que vem à cabeça é *design responsivo e mobile*. Mas elas ajudam também em outro cenário: **acessibilidade**.

Aliás, é sempre bom lembrar que **acessibilidade não é só sobre usuários cegos** (<http://a11yproject.com/posts/myth-accessibility-is-blind-people/>). Há várias categorias, incluindo os que enxergam mas com **dificuldade de ler texto pequeno**. Pode ser uma doença, pode ser uma pessoa mais idosa, ou até um usuário “comum” desconfortável com um texto pequeno.

Para isso, os navegadores têm a capacidade de **zoom nas páginas**. Eu mesmo, apesar de não ter nenhum problema na vista, uso direto: gosto de ler textos com pelo menos 18px de tamanho, então dou zoom em todas as páginas menores que isso.

E, só pra ficar claro, estamos falando agora do *page zoom*, que aumenta tudo na página, usado geralmente no Desktop; não estou falando do gesto de *page scale*, comum no celular, que não afeta a renderização da página, só a visualização do usuário

(veja o tópico 12).

Mas dar zoom na página tem um problema: se você aumentar muito, o **design do site pode quebrar**, principalmente medidas flexíveis como porcentagens e em, tão comuns em design responsivo.



Figura 16.1: Um zoom de 2x deixa o artigo do blog bem estranho. O texto fica grande mas apertado, e a foto fica grande demais sem necessidade.

A parte mais curiosa é que o inimigo desse layout estranho é justo o layout flexível. Se tivesse feito o **layout todo fixo em pixels**, o zoom iria aumentá-lo proporcionalmente e nada quebraria. O ruim só é que o usuário teria que dar scroll horizontal pra ver o site todo.

Media queries ajudam no zoom

Mas a grande sacada é que as **media queries são aplicadas quando o usuário dá zoom**.

Um zoom de 200%, na prática, faz 1px no CSS ser renderizado como 2px na tela. No fim, se você estiver em um notebook de 1280px de tela, a página passaria a renderizar com um *viewport* de 640px. (aliás, há muita semelhança disso com o

tópico sobre resoluções e telas retina 11)

A vantagem do viewport mudar de tamanho de acordo com o zoom é que as **media queries** de `min-width` e `max-width` são aplicadas de acordo com o viewport. Aliás, mais um bom motivo pra não usarmos as media queries `min-device-width` e `max-device-width`, que são relativas ao tamanho da tela e não ao viewport.

As suas media queries criadas originalmente para celulares e tablets começam a ser aplicadas para o usuário com zoom. Seu layout passa a se **adaptar corretamente** sem problema algum.



Figura 16.2: Media queries ajustam o layout no Desktop com zoom de 2x como se estivesse num tablet pequeno. O design fica grande mas bem mais harmônico e proporcional.

Todos os navegadores suportam esse tipo de recurso: Firefox, Chrome, Android, Opera, Internet Explorer e Safari. Só há um bug no WebKit (https://bugs.webkit.org/show_bug.cgi?id=41063) — Chrome, Safari, Android — que faz a media query ser aplicada apenas se o usuário der zoom e depois um *refresh* na página; em todos os outros, a media query é aplicada imediatamente de acordo com o zoom.

Então, se você por acaso não gosta de design responsivo e ainda faz site Desktop e mobile separado, tá aí um motivo pra usar media queries mesmo assim: garantir

acesso de pessoas com visão reduzida. Media queries, no fundo, não tem a ver com mobile mas com *design adaptativo*. E zoom é um tipo de adaptação.

Use media queries com ‘em’ por causa do WebKit

Aliás, falando em WebKit, há outra questão importante. Quando o zoom aumenta, o cálculo dos pixels não afeta as media queries. Isso quer dizer que, se você usou (`max-width: 960px`) no CSS, o WebKit não irá considerar metade desse valor quando o zoom estiver em 200%. Ele deveria fazer isso — como todos os outros browsers fazem —, já que a media query é sobre o **tamanho do viewport e não o tamanho da janela**.

Mas o WebKit consegue reaplicar as media queries no zoom corretamente sim, apenas **precisamos declará-las com em**:

```
@media (max-width: 44em) { ... } /* equivalente a 704px */
@media (max-width: 37em) { ... } /* equivalente a 592px */
```

A conta é simples: é só usar como base o font-size em **16px** e você chega no valor em em. Os browsers WebKit consideram que o font-size base muda conforme o zoom, o que faz sua media query ser aplicada (depois de um refresh, por causa do bug que falei antes).

E não se engane: isso é um **bug do Safari/Chrome/Android e cia**. Usar media queries em em não deveria ser necessário para ganharmos acessibilidade no zoom. **Até o IE funciona direito!** Mas, com o Chrome se tornando o navegador mais usado no mundo e o WebKit dominando completamente o mundo mobile, é bom **fazer sempre suas media queries em em** pra evitar problemas. Pelo menos até corrigirem esse bug.

Media queries de resolução aplicadas no zoom

Uma desvantagem pra quem usa o zoom do browser é que as **imagens costumam ficar pixeladas**. Ela fica menos definida, o que deixa a página feia e até com imagens piores pro deficiente visual entender.

Mas você lembra que temos media queries para lidar com resoluções diferentes (tópico 15)? Elas foram pensadas para serem aplicadas em aparelhos com pixel ratio diferentes (como celulares Retina), mas também **servem para acessibilidade**.

Após um longo debate no W3C (<http://w3-org.9356.n7.nabble.com/Behavior-of-device-pixel-ratio-under-zoom-td6589.html>), todos os navegado-

res concordaram que **alterar o zoom do navegador altera o devicePixelRatio** e, portanto, altera a forma como as media queries `resolution` são avaliadas.

Em outras palavras: se você serve uma imagem de alta resolução (2x) para telas retina, por exemplo, ela será usada quando o usuário der 200% de zoom na página. Excelente.

Enquanto escrevo esse livro, apenas o Firefox implementou isso corretamente. Mas é questão de tempo até os outros navegadores implementarem.

Uma nota sobre a falta de acessibilidade do iOS

O Safari Mobile no iOS não permite mudar o zoom padrão do navegador e ter uma página ajustável como mostrei nesse post. A única opção do usuário é usar o **gesto de page scale** com dois dedos no navegador (*pinch to zoom*) e ficar arrastando a tela de um lado pro outro pra conseguir ver a página toda.

E, claro, isso se o desenvolvedor do site não desabilitou o zoom. Nesse caso, a solução pro usuário é usar o recurso de **zoom nativo do sistema**, que envolve uma série de gestos estranhos e difíceis com 2 e 3 dedos.

Outra nota, sobre zoom vs aumentar a fonte

Nesse tópico, falo do **zoom dos navegadores modernos**, que aumenta a página toda — texto, imagens, CSS etc. Nos navegadores antigos, como IE6 ou IE7, isso não existia e era comum o usuário **aumentar o tamanho da fonte**. Era bem mais simples: só o `font-size` base mudava e a página só aumentaria se você usasse `em` nas medidas CSS.

Aliás, esse era um argumento forte a favor de `em` na época, mas hoje em dia os browsers dão zoom inclusive em medidas com `px` fixos.

Parte III

A Web adaptativa



CAPÍTULO 17

Progressive enhancement e feature detection

Quando falamos de *mobile first*, no fundo, o que falamos é de criar uma base sólida de código portátil e incrementar com recursos avançados para os navegadores mais capazes. Isso é *progressive enhancement* — a melhoria progressiva da página. Esse é um dos pilares do bom web design moderno, fundamental em qualquer projeto front-end mesmo que não envolva mobile.

O *progressive enhancement* é um conceito simples, até fácil de explicar. Aplicá-lo na prática é bem mais complicado. É preciso uma mudança de pensamento e de estratégia. É preciso um excelente planejamento e muito cuidado na execução. Mas é o único caminho para um mundo cheio de browsers e dispositivos diferentes, alguns mais avançados outros mais simples.

Comece identificando as funcionalidades essenciais da aplicação. Programe pensando que essas funções são essenciais em todas as plataformas que o projeto tem que suportar. Faça esse coração da aplicação o mais simples e portátil o possível.

Depois, identifique as funcionalidades a mais que vão incrementar o projeto. Todos aqueles detalhes e facilidades extras que você vai acrescentar pra deixar a aplicação interessante. Aqui, você pode usar os recursos avançados dos navegadores mais modernos, os que até ainda têm suporte bem baixo. Sem problemas: lembre que a base da sua aplicação é sólida e portátil. Esses recursos são incrementos. *Progressive enhancement*.

Um exemplo: você vai fazer um serviço de upload de fotos. A base é super simples: um formulário com um `<input type="file">`. Só. Isso funciona em todos os lugares. Mas, claro, você quer recursos mais modernos e legais.

Pode oferecer upload de múltiplos arquivos ao mesmo tempo, mas que só funciona no IE depois do 10. E colocar um recurso de arrastar e soltar arquivos pra upload automático — que não funciona no IE 9 ou Firefox 3. Pode colocar um preview da imagem sendo enviada na página com canvas. Ou ainda a possibilidade de marcar o local da foto com geolocalização. E adicionar uma barra de progresso de upload bonita, usando sombras e bordas redondas do CSS3. E assim por diante.

Veja como sempre há melhorias possíveis usando APIs mais novas de navegadores novos. File API. Drag and drop. Canvas. Geolocation. Efeitos CSS3. E muito mais. Mas a base é um simples form de upload, que funciona até nos browsers de 15 anos atrás.

Detecção de características

Um truque importante para uma página usar *progressive enhancement* adequadamente é só usar os recursos novos e avançados nos navegadores com suporte para eles. Se fizer o nosso exemplo de upload de fotos todo incrementado e sair rodando no IE6, um monte de erros apareceria por conta de APIs não suportadas.

É preciso um cuidado a mais. Detectar qual navegador suporta qual funcionalidade e só adicionar o recurso nos que são compatíveis. Fazemos o que é chamado de **feature detection**, algo como *detecção de características* dos navegadores.

O nome é bonito mas a ideia é bem simples: testar se o navegador do usuário suporta algum recurso específico e reagir de acordo. Pode ser um simples `if` no JavaScript testando o suporte a alguma API. Por exemplo: se quisermos oferecer o recurso de obter a localização do usuário na nossa aplicação de fotos, podemos testar o suporte a *Geolocation API* em JavaScript facilmente:

```
// testamos se o objeto geolocation existe
if ("geolocation" in navigator) {
```

```
// navegador compatível! usando o recurso:
navigator.geolocation.getCurrentPosition(function(posicao) {
    // ....
});
}
```

Um `if` não muito complicado e, pronto, temos uma página compatível com todo tipo de navegador no melhor estilo *progressive enhancement*.

E podemos, claro, colocar um `else` nesse código. Executar algo nos navegadores que não suportem a *Geolocation API*. Não é estritamente necessário já que a aplicação continua funcional sem esse recurso, dada nossa base sólida e portátil. Mas, às vezes, você quer oferecer uma experiência alternativa para esses navegadores menos capacitados. Por exemplo: deixar o usuário digitar a cidade dele na mão caso a geolocalização não esteja disponível. Esse tipo de estratégia pode ser chamada de *graceful degradation*: usamos um recurso moderno nos browsers bacanas mas a página degrada agradavelmente para os navegadores antigos, sem quebrar ou limitar muito a experiência do usuário.

Modernizr

O mestre da *feature detection* é o projeto **Modernizr**. Nem toda funcionalidade é detectada tão facilmente quanto nosso `if` simples do exemplo de geolocalização. Várias detecções são bem mais complicadas e demandam certo esforço. O papel do *Modernizr* é já cuidar disso e você só se preocupa em escrever um `if` simples.

Um exemplo é se quisermos testar suporte a algum recurso visual do CSS3, como bordas arredondadas. Precisamos, via JavaScript, criar um elemento novo, aplicar a propriedade `border-radius` e depois verificar o resultado na página. E isso pra cada prefixo possível. São algumas linhas de código trabalhoso. Com *Modernizr*?

```
if (Modernizr.borderradius) {
    // temos suporte a bordas arredondadas!
}
```

Muito simples. E mais ainda: cada recurso suportado pelo *Modernizr* pode ser detectado também por CSS além de JavaScript. O framework coloca uma classe no elemento raiz `<html>` indicando suporte ou não a certo recurso. Com isso, você pode fazer certos estilos condicionais:


```
.no-borderradius .menu {  
    border: 1px solid white;  
}  
  
.geolocation .campo-cidade {  
    display: none;  
}
```

A classe `no-borderradius` é colocada quando não há suporte a bordas redondas, senão teríamos a classe `borderradius`. Isso pra qualquer funcionalidade detectável, como `geolocation` e `no-geolocation`.

É um recurso simples mas poderoso para ajudar na adaptação do código da página, tanto CSS quanto JavaScript. E há muitos recursos suportados pelo *Modernizr*, que está sempre crescendo. Você pode detectar certos efeitos de CSS3, animações e transições; APIs como geolocalização, canvas, offline ou drag and drop; input types do HTML5; recursos de áudio e vídeo; suporte a imagens SVG; e mais. Dá pra ver a lista mais atual de recursos do *Modernizr* na documentação: <http://modernizr.com/docs/>

Para usar, é só baixar o arquivo JS do Modernizr e incluir na página. É possível gerar um arquivo específico para seu projeto, apenas com as detecções que você precisar naquela situação, para otimizar o tamanho.

Polyfills

Vários dos recursos modernos do HTML5 / CSS3 já eram usados há muito tempo pelos desenvolvedores na mão. Muitas bibliotecas JavaScript implementavam recursos que, hoje, usamos nativamente nos navegadores. A vantagem, claro, é o recurso mais novo ser uma especificação portátil e já estar integrado ao navegador, deixando a página mais leve.

Hoje, essas bibliotecas JavaScript que simulam recursos do HTML5 podem ter outra utilidade: cobrir o buraco em navegadores velhos que ainda não suportam os recursos novos. Esse tipo de biblioteca é chamada de **polyfill**.

Um bom *polyfill* é completamente transparente para o desenvolvedor. Você usa o recurso como se fosse nativo no navegador, da maneira oficial do HTML5. E esse *polyfill* vai simular o comportamento dos navegadores modernos caso o usuário use um sem suporte.

Por exemplo: o novo atributo `placeholder` do HTML5, que permite colocar um texto de exemplo dentro de inputs e textareas, para facilitar pro usuário preencher o

campo:

```
<input name="telefone" placeholder="(xx) xxxx-xxxx">
```

Telefone:

Isso funciona em todos os navegadores modernos. Nos antigos, será ignorado — o que pode não ser um problema, se esse *placeholder* for apenas um recurso adicional e não essencial. Mas talvez você queira imitar esse comportamento do *placeholder* nos navegadores velhos como Internet Explorer até versão 9. Há vários polyfills como o **Placeholder.js** (<http://jamesallardice.github.com/Placeholders.js/>). É só baixar o arquivo `.js` e apontar na página.

Mas seria ruim baixar esse polyfill em todos os navegadores, até porque a maioria já suporta o recurso nativamente. O ideal seria otimizar isso e só baixar o polyfill caso o navegador em questão não suporte o recurso. É bem fácil de fazer com *Modernizr* que já suporta detecção e carregamento condicional de polyfills:

```
Modernizr.load({  
  test: Modernizr.input.placeholder,  
  nope: '/polyfills/placeholder.js'  
});
```

Esse código executa um teste pra ver o suporte a certa funcionalidade (`Modernizr.input.placeholder`) e carrega um arquivo JavaScript caso não tenha suporte (`nope`). Seria possível carregar outro arquivo no caso positivo usando o parâmetro `yep`.

Existem muitos polyfills para vários recursos do HTML5 e do CSS3. O pessoal do *Modernizr* mantém uma lista no wiki do projeto com várias possibilidades: <https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills>

O futuro da detecção nos navegadores

O *Modernizr* e essa ideia de detectar suporte a funcionalidades do navegador são tão importantes que o CSS incorporou essa técnica. Ainda há pouco suporte nos navegadores, mas uma nova especificação — *CSS Conditional Rules* — traz esse recurso nativamente para a Web.

Existe uma nova regra `@supports` com sintaxe muito parecida com as media queries, mas para testar se o navegador suporta certo recurso do CSS. Por exemplo, pra ver se as bordas redondas são suportadas, poderemos fazer só:

```
@supports (border-radius: 1em) {  
    // regras CSS caso haja bordas redondas  
}
```

O navegador só vai avaliar o CSS dentro do bloco caso a regra testada seja suportada. Para saber mais: <https://developer.mozilla.org/docs/CSS/@supports>

Mobile e design adaptativo

O que todo esse papo sobre *Modernizr* faz num livro sobre *Web Mobile*? Eu falei na introdução que o título do livro é meio limitado. Precisamos é encarar o *Web Design moderno* como um todo; apenas, agora, incluindo uma série de novos dispositivos na mesa. E a questão toda é que mais dispositivos trazem mais navegadores diferentes, mais contextos diferentes e mais variações entre os casos de uso da sua página.

Quando falamos do *web design responsivo* no sentido clássico de quando o termo surgiu é fácil acabar focando demais nos tamanhos das telas. Media queries, grids flexíveis, imagens responsivas. Tudo isso lembra muito o foco nos tamanhos diferentes de telas. Mas há muitas outras diferenças entre os aparelhos e os navegadores.

Hoje fala-se muito de **design adaptativo**, um conceito mais amplo do que só deixar a página responsiva. É falar sobre sua página e seu código se adaptarem a diferentes situações, levando em conta tamanho de tela, mas também capacidades dos navegadores, características do hardware do aparelho e até o contexto de uso do visitante. Vamos nos aprofundar mais nisso no tópico 20.

Detectar as funcionalidades de cada browser como vimos aqui nos ajuda a implementar o *design adaptativo* e tem tudo a ver com design responsivo e suporte a vários dispositivos e navegadores. Você pode detectar suporte a eventos *touch* e várias outras características comuns de celulares e tablets.

Foi-se o tempo em que pensávamos em criar uma página com funcionalidades bem definidas e a maior preocupação era se o browser X ia rodar ou não a página. Hoje, no web design moderno, as palavras chave são **adaptação e flexibilidade**. A mesma página executar de forma diferente dependendo do browser, e se adaptar a novas condições. É bem mais desafiador, claro.

CAPÍTULO 18

RESS — Design responsivos com componentes no lado do servidor

O tipo de detecção de funcionalidades que vimos no tópico anterior (17) é executado direto do navegador do usuário. Tanto *Modernizr* quanto o futuro `@supports` são *client-side*.

A grande vantagem desse tipo de detecção é ter bastante precisão nos resultados. Apesar de não serem infalíveis, as detecções são feitas observando diretamente o navegador em que estamos executando a página. É possível captar muitas informações e detectar muita coisa diferente.

Mas detecções *client-side* têm suas deficiências. Por rodarem no navegador, gastam recursos da máquina do usuário e sobrecarregam um pouco mais a página. Mais ainda: seu código vai tomar decisões com base nas detecções direto no browser, enquanto a página executa. Isso quer dizer que você pode acabar carregando códigos que nem são executados se você descobrir que o navegador em questão não suporta aquela funcionalidade.

Ou ainda, por ser client-side, a detecção pode até identificar suporte a certa funcionalidade, mas pode ser tarde demais para usá-la. Um exemplo clássico é com formatos de imagens. Você quer usar um logotipo em SVG mas com uma versão PNG para os browsers velhos. Na hora de escrever a tag, você provavelmente vai escrever `` que tem mais suporte. Aí detecta o suporte a SVG com *Modernizr* e troca, via JavaScript, o `src` da imagem para `logo.svg`. Isso funciona, mas o navegador vai acabar baixando as duas versões (PNG e SVG), gastando recursos.

Nesse cenário, detectar o suporte a SVG no *client-side* é tarde demais. A imagem PNG já foi baixada e trocar por SVG via JavaScript vai causar mais problemas. O ideal era que o HTML da página já referenciasse o formato adequado pra cada browser, detectando o suporte direto no servidor.

RESS

O termo **RESS** quer dizer **Responsive Design + Server-Side components**, ou *Design Responsivo + componentes do lado do servidor*. Foi cunhado por Luke Wroblewski, um desenvolvedor mobile de destaque, em um post famoso (<http://www.lukew.com/ff/entry.asp?1392>).

A ideia é misturar tudo que vimos até agora de *detecção e adaptação* das páginas com lógica executada também no servidor. Em vez de rodar toda lógica no navegador (*client-side*), podemos executar certas otimizações e detecções antes, direto no servidor (*server-side*) e já enviar um HTML mais apropriado para cada navegador.

Essa ideia, claro, depende de você ter um servidor e conhecimentos de programação *server-side*. Se estiver fazendo um projeto puramente front-end ou *packaged apps* com HTML5, não vai resolver muito. Mas a maioria das webapps tem um lado no servidor, rodando em alguma linguagem como PHP, Java, .Net, Ruby, Python, ASP etc. Nesse caso, RESS pode ser bem útil.

Veja o caso de mandar o tipo de imagem correto pra cada navegador. Vou usar PHP para dar um exemplo de código RESS, mas poderia ser qualquer linguagem de servidor, claro.

```
<div class="logo">
  <?php if ($suporta_svg) { ?>
    
  <?php } else { ?>
    
```

```
<?php } ?>  
</div>
```

Esse código mistura um pouco de PHP no HTML pra fazer um `if` simples. Testo se a variável `$suporta_svg` é verdadeira e aí envio a tag `img` carregando SVG; se não, vai a versão em PNG. A ideia é que cada browser receba uma tag `img` simples e apropriada para as funcionalidades que suporta. Não precisaremos rodar detecção com *Modernizr* nem nenhum truque de JavaScript.

Claro que a questão toda é: como determinar no servidor que o navegador suporta ou não SVG? Ou, mais amplo, como detectar no servidor qualquer característica que temos interesse em testar?

Bancos de dados de dispositivos

As únicas informações que o servidor tem acesso direto são os cabeçalhos HTTP, o que inclui o *User-Agent*. Ele é um cabeçalho que vem nas requisições e identifica o navegador, sua versão, sistema operacional e algumas outras poucas características. Se você nunca viu o *User-Agent*, pode ver o do seu navegador acessando, em JavaScript, `navigator.userAgent`. O meu Chrome que estou usando enquanto escrevo esse livro se identifica como:

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_5) AppleWebKit/537.35  
(KHTML, like Gecko) Chrome/27.0.1448.0 Safari/537.35
```

É um texto confuso, mas uma coisa é certa: aí não está falando quais tipos de formatos de imagens são suportados, e nem as centenas de outras características que gostaríamos de testar. Mas é o que temos à mão para identificar um browser e, depois, tentar supor suas capacidades.

O que as pessoas fazem é criar **gigantescos bancos de dados de User-Agents**. Cada browser é testado para todo tipo de funcionalidade importante e associado a seu *User-Agent*. Aí quando chega uma requisição no servidor, você pega o *User-Agent* e procura nesse banco de dados as características do navegador.

Existem soluções comerciais com bancos de dados imensos que catalogam praticamente todo tipo de dispositivo do mundo. As mais famosas são **WURFL** (<http://www.scientiamobile.com>), **DeviceAtlas** (<https://deviceatlas.com>) e **51degrees** (<http://51degrees.mobi>). São produtos pagos que são oferecidos geralmente em dois sabores. Uma versão que você instala no seu servidor, e outra versão *cloud* na qual você chama uma API e busca no servidor central deles. A vantagem da solução central é

a facilidade de uso e não ter a necessidade de atualizar o banco de dados toda hora; a desvantagem é ser um pouco mais lento.

O WURFL era um projeto opensource até 2012, com o banco de dados publicamente acessível (<http://wurfl.sf.net>). Mas eles pararam de atualizar o projeto e passaram a oferecer a versão comercial. Como o valor de um banco de dados de dispositivos está justamente em mantê-lo sempre atualizado, essa versão opensource não é muito útil, apesar de ainda estar disponível. Existe um fork da última versão livre do WURFL, chamada *OpenDDR* (<https://github.com/OpenDDRdotORG>) que tenta manter o banco de dados atualizado.

Um exemplo com WURFL em PHP para detectar suporte a SVG, como queríamos:

```
// cria o objeto de detecção
$client = new WurflCloud_Client_Client(...);
$client->detectDevice();

// descobre se suporta SVG
$suporta_svg = $client->getDeviceCapability('svgt_1_1');

// ... envia a <img> no formato certo ...
```

Mas é possível detectar muito mais coisas. Por exemplo, otimizar o layout da página para touchscreens:

```
if ($client->getDeviceCapability('pointing_method') == 'touchscreen') {
    // ... otimiza para telas touch ...
}
```

É possível pegar muito mais dados, como tamanho físico do aparelho, resolução da tela, suporte a certos recursos do HTML5 e mais. Uma lista das capacidades detectáveis pelo WURL pode ser encontrada no site deles: <http://www.scientiamobile.com/wurflCapability/>. Os outros concorrentes têm funcionalidades similares.

Uma alternativa é tentar detectar apenas o tipo de dispositivo e não cada possível característica. Muitos sites detectam navegadores mobile e fazem um redirect para um site mobile específico. Ou mesmo se você não for fazer o redirect, pode fazer algum tipo de customização ao saber só o tipo do browser. Uma solução é o <http://detectmobilebrowsers.com> que tem uma expressão regular gigantesca em várias linguagens de programação que só diz se é um smartphone ou não.

Misturando detecção client-side com server-side

Sistemas de detecção baseados no *User-Agent* têm alguns problemas também. Você precisa sempre deixá-lo atualizado, o que é trabalhoso. Novos dispositivos podem demorar para aparecer no catálogo. Além disso, vários navegadores não se identificam unicamente via *User-Agent*. Por exemplo, meu navegador pessoal favorito no Android é o *Dolphin* mas se identifica como o navegador padrão do Android, apesar de ser bem mais avançado que ele. O Opera também é famoso por permitir facilmente o uso de *User-Agent* falso. O iPad Mini e o iPad normal têm o mesmo *User-Agent*, então não é possível afirmar o tamanho físico do aparelho com certeza.

Usando o *User-Agent* então temos o problema de identificar erradamente o dispositivo.

Mas, mesmo quando acertamos, podem existir bugs nos bancos de dados. Testando essas ferramentas enquanto escrevo o livro, 2 delas indicaram que meu celular (Galaxy S II) não suporta imagens SVG. Testei tanto o Chrome, quanto o Dolphin e o navegador padrão do Android 4.1 — todos eles suportam sim SVG apesar de estarem marcados negativamente nos bancos de dados. Uma detecção *client-side* com *Modernizr* não teria errado nesse cenário.

Há então quem defenda uma mistura de detecção *client-side* com *server-side*. Você roda o *Modernizr* na primeira visita do usuário detectando tudo que interesse *client-side*. Ai salva essas informações no servidor e pode usá-las depois no *server-side* e para não precisar detectar tudo de novo em uma nova visita. Se quiser, ainda é possível gravar essas informações num banco de dados associado ao *User-Agent* do navegador e usar esse resultado pré-computado com navegadores de outros usuários que sejam iguais (apesar que, como vimos antes, navegadores diferentes podem usar o mesmo *User-Agent*).

Uma ferramenta gratuita que faz esse tipo de detecção misturando *client* e *server* é a **Detector** (<http://detector.dmolsen.com/>). Quer saber se o navegador tem suporte a geolocalização?

```
include("detector.php");

if ($ua->geolocation) {
    // ... tem geolocation API! ...
}
```

Implementar uma solução assim na mão também não é muito complicado. Você roda o *Modernizr* e guarda os resultados num cookie que depois será lido pelo servidor. Vamos supor que queremos detectar suporte a SVG e a geolocalização:


```
var suporte = '';

if (Modernizr.svg) {
    suporte += 'svg,'
}
if (Modernizr.geolocation) {
    suporte += 'geolocation,'
}

document.cookie = 'modernizr=' + suporte;
```

O código anterior cria um cookie chamado `modernizr` cujo valor é uma sequência de termos indicando suporte a certa funcionalidade (no exemplo, `svg` e `geolocation`).

No servidor, podemos pegar o valor do cookie e checar suporte a algum dos pontos detectados. Voltando ao nosso exemplo de enviar uma imagem SVG se houver suporte:

```
$capacidades = explode(",", $_COOKIE['modernizr']);
if (in_array("svg", $capacidades)) {
    // ... envia imagem SVG! ...
}
```

Esse código PHP pega o valor do cookie `modernizr` e quebra em um array com todas as capacidades detectadas no JavaScript. Aí só verificamos se existe suporte a `svg` antes de mandar a imagem nesse formato. Bem simples. E dá pra estender pra muitas outras coisas, como mandar o tamanho da tela, se é retina ou não e muito mais.

Há algumas limitações nessa abordagem, porém. Como precisamos rodar o JavaScript primeiro para detectar as capacidades do navegador, o servidor não tem essas informações no primeiro request. As abordagens possíveis são enviar uma versão padrão no primeiro request, ou forçar um reload na página para o request ser refeito.

Lembre ainda que não conseguimos detectar algumas coisas *client-side*, em especial as características físicas do aparelho — tamanho, DPI, tipo de hardware etc. Como esse tipo de informação é facilmente encontrada nos bancos de dados de dispositivos que vimos antes, você também pode optar por uma abordagem mista.

Muita coisa, porém, ainda é indetectável até hoje. Num projeto recente, eu queria identificar o suporte a URLs para disparar ligações telefônicas (`tel:`). Não há ainda

como detectar suporte nem em *client-side* nem em *server-side*.

O que fazer com RESS

Como vimos, são portanto várias as maneiras de se detectar no servidor qual é o aparelho e quais as capacidades do navegador. Mas o que fazer de útil com essas informações?

O primeiro tipo de otimização possível é ajustar os componentes da página. Vimos como enviar uma imagem SVG para navegadores compatíveis e PNG para os demais. Poderíamos fazer a mesma coisa para imagens WEBP — um formato novo bem mais otimizado que JPEG, mas ainda com pouco suporte.

Outra possibilidade seria incluir os arquivos de JavaScript das bibliotecas e polyfills necessários apenas para aquele navegador específico. Ou mandar imagens em alta definição para telas retina. Ou ainda enviar um CSS sem media queries para navegadores antigos que não suportem, como o IE8.

Além disso, podemos alterar o design da página. Embora as media queries façam boa parte do trabalho de reorganizar o design para diferentes tamanhos de tela, há várias limitações. Por mais criativo que sejamos no CSS, até hoje a ordem dos elementos no HTML tem um forte impacto no design final da página. Não é simples, por exemplo, colocar seu `<header>` no fim do arquivo HTML e fazê-lo aparecer no topo no design apenas com CSS.

Com RESS, podemos alterar a ordem dos elementos no HTML (*source order*) como for mais conveniente. Uma possibilidade é colocar o menu no fim da página em telas pequenas mobile, e colocá-lo no topo no Desktop e telas grandes. E muitos outros rearranjos no HTML que depois facilitam a implementação do design responsivo no CSS.

Mais uma ideia: colocar um widget do Google Maps no Desktop mas trocar por um link simples no mobile. Isso faz a página ficar mais leve e ainda integra com a aplicação nativa de mapas.

São muitas as possibilidades com RESS. Mas repare que os usos mais comuns são para ajustes pontuais. A base da página é a mesma para todo dispositivo e só ajustamos detalhes com base nas características de cada usuário. Não são mudanças radicais como fazer dois sites totalmente diferentes.

Um detalhe importante sobre RESS: cuidado com caches em proxys e intermediários. Se você está servindo na mesma URL conteúdos diferentes dependendo do visitante, precisa indicar isso nos headers senão é possível que alguém faça cache

da versão Desktop e sirva para um smartphone essa versão errada cacheada (e vice-versa). O jeito correto é adicionar no cabeçalho `Vary` na resposta do servidor. Se for usar um banco de dados baseado em *User-Agent*, use `Vary: User-Agent`; se for usar uma solução baseada em cookies, use `Vary: Cookie`.

Como agir se o RESS falhar

RESS é bem interessante mas muita coisa pode dar errado. Vimos que o banco de dados pode errar na detecção, o navegador pode se identificar erroneamente, a base pode estar desatualizada etc. O que fazer se as coisas derem errado?

O ideal é manter as alterações do RESS confinadas a pequenos ajustes e detalhes e tentar garantir que a experiência do usuário no final seja aceitável e não completamente errada.

Pegue o exemplo que citei antes de colocar um link para o mapa nativo ao invés do widget do Google Maps. Tanto no iOS quanto no Android, um link para <http://maps.google.com> vai abrir a aplicação nativa, por isso trocamos o widget por esse link. Mas e se a detecção errar e acabarmos enviando a versão com link para uma plataforma sem Google Maps nativo? Ele não vai ver o widget, mas o link funciona normalmente, abrindo o site mobile do Google Maps. Uma experiência aceitável.

No outro cenário de enviar o SVG para navegadores compatíveis. Se a detecção falhar e achar que meu navegador não suporta SVG (como aconteceu no meu celular), mandamos o PNG e está tudo bem, é aceitável. Mas e se falhar ao contrário? Achar que tem SVG mas não tem? Nesse caso, o navegador vai mostrar um erro nas imagens e o site fica inutilizável. Péssimo. Mas podemos melhorar: além de rodar a detecção no servidor, faça um JavaScript que detecte erros no navegador e carregue o PNG por cima. Seria bem simples:

```

```

Se der erro, o usuário vai ter gastado 2 requests, mas a experiência final é aceitável, o site continua usável. Lembre: é só uma estratégia para o cenário do RESS falhar na detecção.

E aquele exemplo de trocarmos o menu de lugar no HTML no mobile e no Desktop. Faça o site ser usável em todas as combinações. Se eu acabar mandando o HTML no lugar errado por falha na detecção, faça o CSS mostrar de maneira usável em ambos lugares, tanto no Desktop quanto no mobile.

É a ideia de **programar defensivamente**, prevendo que coisas podem dar errado.

Ainda mais nesse cenário de detecção com base em *User-Agent* que pode falhar com certa frequência. Se o *server* errar, faça com que o *client* consiga se recuperar.

CAPÍTULO 19

O complicado cenário das imagens responsivas

Uma parte bem complicada de um bom design responsivo é resolver como lidar com as imagens. Elas costumam ser feitas de um número fixo de pixels e são grandes vilãs de um design fluído. Podemos, porém, colocar seu tamanho em porcentagens (`width:100` ou `max-width:100`) e a imagem irá fluir junto com o design.

O problema é que colocar porcentagens fará a imagem aumentar ou diminuir de acordo com o tamanho da tela. E imagine uma imagem de 300px sendo esticada a 100% de uma tela de 1280px. Fica bem ruim. Por isso, em muitos cenários, precisamos lidar com imagens diferentes e otimizadas pra cada cenário.

As pessoas falam de *imagens responsivas* para descrever o problema de servir imagens diferentes para cenários diferentes. Mas que cenários seriam esses? Alguns:

- Não servir imagens gigantes da versão Desktop para a versão mobile pequena;
- Imagens bem definidas em telas retina de alta resolução;

- Qualidade nas imagens quando damos zoom tanto no Desktop quanto no celular;
- Permitir imagens totalmente diferentes no Desktop e no mobile. Um caso é que você pode preferir um recorte diferente quando a imagem for pequena no celular, pra focar no elemento principal. Pessoal chama isso de *direção de arte* (*art direction*).

São vários cenários onde a imagem deve responder ao contexto, ou seja, deve se adaptar. E as boas e velhas imagens de antigamente não fazem nenhum desses truques.

Se analisar bem os cenários que descrevi antes, chegamos na verdade em 3 objetivos principais:

- economizar bytes enviando a imagem do tamanho certo pra cada tela;
- obter qualidade visual na renderização da imagem de acordo com a resolução;
- usar imagens de conteúdos diferentes para adaptar à necessidades de design.

Vamos focar primeiro nos dois primeiros objetivos e depois falar um pouco do terceiro.

Não use imagens, prefira CSS

É óbvio: se não quer se complicar com imagens responsivas, a primeira solução é evitar usar imagens! Não estou falando de fazer um design mais simples sem imagem nenhuma (apesar de preferir designs mais cleans), mas de recriar aspectos visuais do design com outras técnicas sem ser imagens.

Em especial, usar efeitos CSS3. Antigamente, era comum criar imagens quando precisávamos de bordas redondas, sombras, gradientes e vários outros efeitos que, hoje, são facilmente recriáveis via código CSS3. A grande vantagem de preferir CSS a imagens pra esses efeitos é que o CSS é renderizado no navegador do cliente, ajustado para tamanho da tela e resolução automaticamente.

Escrever CSS3 também facilita manutenções futuras — mudar a cor de um gradiente é só editar o código, não precisa ir no Photoshop, editar, exportar imagem nova etc. Usar código CSS também deixa o download mais leve — é só uma propriedade texto que descreve o efeito, em vez de um request para uma nova imagem potencialmente grande.

Um ponto negativo a se pesar é a performance na renderização desses efeitos. Como é o navegador quem desenha o efeito, há um gasto na máquina do cliente. Os navegadores estão cada dia mais rápidos e os hardwares, melhores; mas, mesmo assim, se você abusar colocando vários gradientes e sombras gigantes na mesma página, vai sentir um impacto na velocidade da renderização. A dica é usar o CSS e observar se seu uso está sendo benéfico ou não.

Texto é texto

Outra forma de evitar imagens é usando texto. Texto é renderizado muito bem nos navegadores e também se adapta muito bem a telas de alta resolução. Além disso, o texto é fluído no HTML e pode ser reconfigurado à vontade via CSS para diferentes cenários. Você pode facilmente usar CSS e media queries para ajustar o texto e seu design para diferentes tamanhos de telas e layouts responsivos distintos.

Uma prática muito ruim mas muito comum antigamente era colocar texto dentro de imagens e exportar tudo junto no Photoshop. Há muitas desvantagens nessa abordagem. Dar manutenção é bem trabalhoso, o Google não indexa esse texto (SEO) e ele fica borrado em telas retina. Além disso, o texto fica fixo na posição dele na imagem e não pode ser adaptado com media queries em um design fluído.

Hoje, com excelente suporte nos navegadores para fontes customizadas com `@font-face`, o recomendado é sempre colocar texto como texto no HTML, jamais como HTML. Fica mais leve, melhor renderizado e te dá flexibilidade para manipulação no CSS.

Icon fonts

A ideia de resgatar o uso de texto puro e fontes customizadas está tão forte que já se espalhou até para ícones. Uma prática com bastante adoção ultimamente é o uso de **fontes de ícones**, as *icon fonts*. Como podemos carregar fontes customizadas, podemos gerar uma fonte cheia de ícones ao invés de caracteres e usá-los no design.

Há muitas fontes no mercado compostas apenas de ícones dos mais diversos. Há também ferramentas que permitem que você transforme seus próprios ícones em fontes. Uma ferramenta que eu gosto bastante é a **IcoMoon** (<http://icomoon.io/app>), um serviço online que gera fontes de ícones customizadas pra você. Você pode escolher os ícones que deseja dentro do imenso catálogo deles e até importar seus próprios ícones. No final, ele gera os arquivos de fontes otimizados só com ícones que você escolheu e ainda te dá o CSS de exemplo para usar na página.

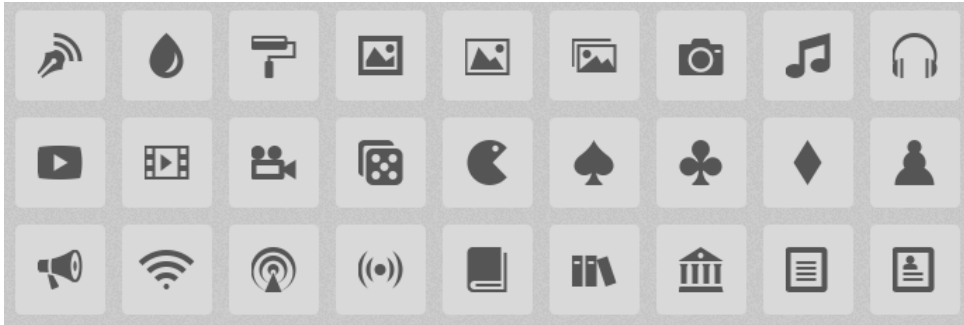


Figura 19.1: Alguns poucos ícones do IcoMoon, dentre as centenas disponíveis.

Usar ícones como fontes tem muitas vantagens:

- As fontes são renderizadas muito bem nas telas de alta resolução e não ficam borradas;
- Um arquivo só de fonte pode conter vários ícones juntos, otimizando os requests; é quase como criar uma sprite de imagens;
- Você pode customizar alguns aspectos via CSS, como cor, tamanho, bordas e sombras usando um único ícone.

A desvantagem maior das fontes de ícones é sua simplicidade. Os desenhos precisam ser um vetor de um único path, uma única cor. Não é possível adicionar desenhos complexos, com camadas etc. Você pode usar mais de um ícone pra cada parte do desenho e depois compor a imagem final usando vários ícones ao mesmo tempo, mas isso já é um pouco mais trabalhoso.

Outra questão que é importante notar é a diferença de renderização das fontes entre sistemas operacionais e entre navegadores. Os designers mais atentos vão perceber diferenças na forma como cada um faz seu *antialiasing* e até de posicionamento. Se for usar icon fonts, esteja preparado para aceitar pequenas diferenças de renderização.

A volta do SVG

A grande vantagem de se usar as fontes para texto e ícones na verdade é o fato de serem formatos **vetoriais** ao invés dos formatos clássicos da Web baseados em pixels

como PNG ou JPEG. Um ícone vetorial é gravado como coordenadas de pontos e formas geométricas ao invés de um pixels individuais, por isso pode ser renderizado em vários tamanhos sem problemas de ficar borrado.

O auge das imagens vetoriais na Web é o formato **SVG**. Nascido no último milênio, viveu meio esquecido por vários anos. Mas, hoje, é estrela novamente na Web por permitir imagens independentes de resolução. Podemos usar SVG para ícones, logos e desenhos bem completos. Todos os programas gráficos vetoriais (Illustrator, Corel, Inkscape) exportam SVG para uso na Web.

A maioria dos navegadores já suporta SVG há um bom tempo — as maiores exceções são IE8 e anteriores, e Android 2.x (<http://caniuse.com/#search=SVG>). Mas lembre que você pode facilmente detectar suporte no navegador e oferecer um PNG para os navegadores mais antigos.

O SVG é um formato texto baseado em XML. Você pode abrir o arquivo e editá-lo, mas geralmente você vai preferir desenhar num programa gráfico. Para usá-lo na página, há algumas opções. A mais comum é usar direto na tag `img`:

```

```

Usando a tag `img`, não conseguimos fazer um fallback de forma de fácil para navegadores sem suporte. O jeito comum é detectar suporte com *Modernizr* e trocar o `src` para apontar para o PNG caso não tenha suporte, mas isso vai causar o download de ambos os arquivos nesses navegadores. Outra solução é RESS, como discutimos no capítulo 18.

Também é possível usar como `background-image` no CSS, o que torna bem fácil implementar o mecanismo de fallback pra PNG. O *Modernizr* colocar uma classe `svg` ou `no-svg` para indicar o suporte a esse formato. Então podemos fazer algo como:

```
.fundo {  
    background-image: url(fundo.png);  
}  
.svg .fundo {  
    background-image: url(fundo.svg);  
}
```

No código anterior, indicamos o arquivo SVG caso haja suporte, se não, o PNG é carregado. A vantagem é a simplicidade da solução e o fato do navegador não baixar os dois arquivos em nenhuma circunstância.

Outra possibilidade acrescentada no HTML5 é a de usar SVG direto no meio no HTML, já que ambos são formatos baseados em tags e compatíveis. Algo assim:

```
<html>
<body>
  <h1>Minha página!</h1>
  <p>A imagem abaixo é um triângulo em SVG.</p>

  <svg>
    <polygon points="0,100 0,0 100,0" />
  </svg>

</body>
</html>
```

O código anterior é bem simples e mostra um triângulo desenhado com a tag `<polygon>` do SVG. Repare como usamos o elemento `<svg>` no meio do HTML. Isso é permitido e ajuda a economizar um request. Além disso, usando dessa maneira, você pode usar JavaScript para manipular os elementos, animá-los etc. Pode também usar CSS para manipular as características do desenho.

SVG é uma ferramenta bem poderosa para evitar lidar com a inflexibilidade das imagens comuns em PNG ou JPEG. Para desenhos sem muita complexidade é comum até que o tamanho do arquivo SVG fique menor que uma versão idêntica em PNG. Cuidado apenas com desenhos SVG muito complexos, já que tudo é renderizado no navegador. Desenhos complexos geram arquivos de tamanho bem grande e demandam bastante do navegador para renderizar.

Uma última dica sobre SVG é exportá-lo usando seu programa de desenho mas depois otimizá-lo com uma ferramenta apropriada que vai remover coisas desnecessárias do código. Duas ferramentas famosas: **svgo** (<https://github.com/svg/svgo>) e **scour** (<http://www.codedread.com/scour/>). Por fim, não esqueça que o SVG é um formato texto e, portanto, pode ter seu tamanho de download bastante reduzido ao usar GZIP no servidor.

Imagens diferentes com media queries

Nem todas as imagens são desenhos vetoriais que você pode trocar por SVG ou icon fonts. Temos também as fotografias, geralmente salvas em JPEG. E algumas vezes vamos acabar preferindo PNG ao invés de SVG também. A questão é: em vários cenários ainda teremos que lidar com imagens em formatos clássicos (PNG e JPEG) e queremos um certo nível de adaptações.

O primeiro cenário a se considerar é o de enviar as imagens nos tamanhos mais apropriados para cada tamanho de tela. Imagine um site onde queiramos uma ima-

gem grandona que ocupe a largura toda da tela. Criamos um JPG de 960px para Desktop mas parece desperdício enviar esse mesmo arquivo para um celular de só 320px de largura. Queremos arquivos diferentes para cada contexto.

Com media queries é bem fácil resolver isso se estivermos lidando com background-image:

```
.banner {  
    background-image: url(banner-mobile.jpg);  
}  
  
@media (min-width: 400px) {  
    .banner {  
        background-image: url(banner-medio.jpg);  
    }  
}  
  
@media (min-width: 700px) {  
    .banner {  
        background-image: url(banner-grande.jpg);  
    }  
}
```

Você vai ter que escolher quantas versões de cada imagem vai disponibilizar e para quais tamanhos de tela. Dá trabalho mas não há muito jeito. É isso ou servir a mesma imagem pra todo mundo.

Mas o pior caso é quando lidamos com a tag . Não conseguimos hoje o mesmo tipo de flexibilidade do CSS com media queries. Uma solução seria não usar as tags e trocar tudo para imagens via CSS, mas isso pode não ser muito sustentável.

Existem algumas soluções em JavaScript que trocam o src da imagem com base em certa condição. O problema de toda solução JavaScript é ser mais lento que uma solução puramente HTML ou com CSS. Os navegadores de hoje têm diversas otimizações para lidar com imagens comuns (no HTML ou no CSS) e quando seguramos o carregamento para ser disparado pelo JS, atrapalhamos várias dessas otimizações. Isso sem contar soluções JS mais ingênuas que trocam o src mas acabam disparando o download de ambas as versões da imagem, o que é pior ainda.

Já existem soluções para imagens responsivas na especificação do HTML5. A mais simples é um novo atributo srcset na tag que permite listar mais de

um arquivo ao mesmo tempo e deixar o navegador escolher o mais apropriado naquela situação. Cada arquivo pode ser listado com características como tamanho do viewport ou densidade de pixels. Veja um exemplo:

```
<img srcset="mobile.jpg 400w, medio.jpg 700px, grande.jpg">
```

No código acima, 400w indica que a imagem `mobile.jpg` deve ser carregada no caso do viewport ser no máximo 400 pixels — é uma versão simplificada da media query `max-width`. Repare como podemos listar vários arquivos e descrever as condições em que cada uma deve ser carregada.

Há também uma versão equivalente do `srcset` no CSS através da função `image-set()` que pode ser usada nos backgrounds:

```
.banner {
  background: image-set(url(mobile.jpg) 400w, url(grande.jpg));
}
```

Enquanto escrevo esse livro, o suporte nos navegadores ainda é muito baixo, infelizmente. Além das soluções do `image-set` e `srcset`, há uma outra especificação criando um novo elemento `<picture>` com suporte total a media queries e baseado no formato atual das tags `<video>` e `<audio>`. Ela permitirá algo assim:

```
<picture>
  <source src="mobile.jpg">
  <source src="grande.jpg" media="(min-width: 700px)">
  
</picture>
```

Por ser uma media query comum, é possível todo tipo de customização. Mas é uma sintaxe mais verbosa e mais trabalhosa. Ainda há muita discussão sobre essa nova tag, se ela realmente vai virar oficial ou não.

Por enquanto, há um polyfill em JavaScript do Scott Jehl chamado **Picturefill** (<https://github.com/scottjehl/picturefill>). Ele usa uma sintaxe baseada na especificação mas usando `div`s já que os novos elementos não existem ainda. O código fica mais ou menos:

```
<div data-picture>
  <div data-src="mobile.jpg"></div>
  <div data-src="grande.jpg" data-media="(min-width: 700px)"></div>
  <noscript></noscript>
</div>
```

É uma solução mais verbosa ainda e poluída, além de ser baseada em JavaScript o que deixa tudo mais lento. Eu pessoalmente tenho usado a estratégia do CSS com media queries, sem nada de JavaScript. É uma solução mais limpa e simples para os navegadores de hoje. Mas fique ligado nas evoluções ao redor das maneiras oficiais do futuro (`srcset` e `image-set()`).

E todas essas soluções para servir imagens diferentes para tamanhos de telas distintos também resolvem a questão da *direção de arte*. É aquele cenário onde queremos servir imagens diferentes dependendo do tamanho da tela, geralmente fotografias recortadas com ênfases específicas em cada tamanho. A ideia é que uma foto pequena no celular pode exigir uma certa cena, diferente da foto grande no Desktop.

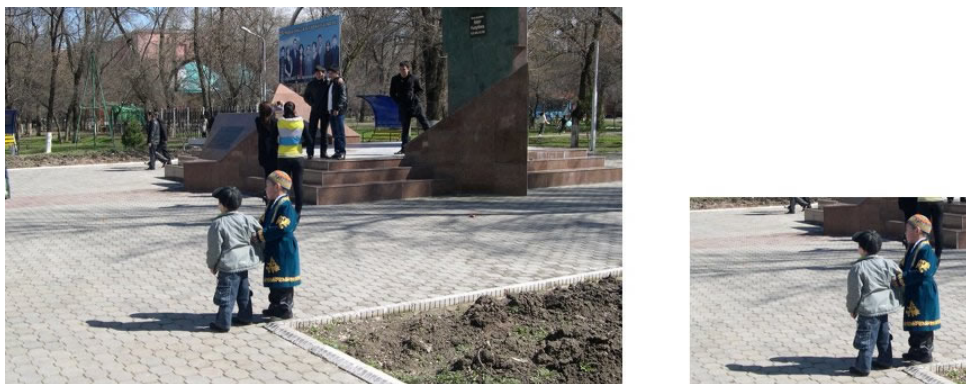


Figura 19.2: Uma foto em tamanho pequeno com um recorte diferente.

Estratégias para telas retina

Suportar imagens para telas retina é um caso particular. A questão é que imagens comuns ficam bem feias em telas com *pixel ratio* maior que 1. Lembre que uma tela retina, por exemplo, tem *pixel ratio* 2x. Logo, uma imagem de 100px por 100px vai ocupar 200px por 200px nos pixels físicos. O navegador aumenta 1px da imagem pra ocupar a área de 4px físicos, deixando a imagem bem borrada.

O problema fica mais acentuado ainda em páginas com vários elementos CSS e texto. Isso porque, como vimos, texto, CSS, icon fonts e SVG são renderizados lindamente em telas retina. Aí as antigas imagens PNG e JPG ficam bem feias em comparação, por acabarem escalando os pixels.

A solução mais óbvia nós já discutimos: prefira recursos vetoriais e evite imagens como PNG ou JPG. Imagens vetoriais, fontes e CSS são renderizados de acordo com

a densidade de pixels da tela, então se adaptam automaticamente a aparelhos retina ou não retina.

Mas se você precisar usar JPG ou PNG, uma estratégia é servir imagens diferentes dependendo do *pixel ratio*. Usando media queries no CSS para imagens de background, faremos:

```
.painel {
    background: url(fundo-100-por-100.jpg);
}

@media (-webkit-min-device-pixel-ratio: 1.5),
    (min-resolution: 144dpi) {

    .painel {
        background: url(fundo-200-por-200.jpg);
        background-size: 100px 100px;
    }
}
```

Estamos usando as media queries de resolução como discutimos no tópico 15. Além disso, repare na propriedade `background-size`. Ela é importantíssima, pois diz para o navegador renderizar a imagem com metade do tamanho dela, já que estamos servindo uma com o dobro da resolução. Sem ela, a imagem ficaria gigante na tela, com 200px.

Além disso, as sintaxes dos futuros `srcset` e `image-set` também suportam configurações de densidades de pixels:

```
<img srcset="painel-100.jpg, painel-200.jpg 2x" width="100" height="100">
```

Compressive images

As soluções anteriores pressupõem que queremos imagens diferentes para cada densidade de pixels. Pode parecer estranho, mas muita gente está defendendo a ideia de servir uma imagem só pra todo mundo — retina e não retina.

Primeiro, tenha em mente que telas retina são o futuro. Não ignore-as agora pois em breve elas serão a regra e não a exceção. Nos celulares e tablets isso já aconteceu. Os notebooks e desktops já começaram. Então queremos que nosso site de hoje esteja preparado para retina sempre.

A questão a se pensar é: e se servirmos **apenas as versões retina das imagens**? Pode parecer meio loucura. Vamos gastar recursos à toa. Mas pode não ser bem assim.

No caso dos JPEGs, uma estratégia que tem sido bem aceita é a **compressive images**. É gerar uma imagem JPEG grande, retina, com o dobro da resolução, mas com qualidade bem baixa. Na hora de exportar a imagem no seu editor favorito, podemos escolher a qualidade — um número de 1 a 100. A ideia é gerar nossa imagem retina com **qualidade 30** por exemplo, um número bem baixo. Aí incluímos essa imagem na página declarando a metade do tamanho, como vimos antes.

Para surpresa de muitos, essa imagem fica com qualidade visual muito boa tanto em telas comuns como em telas retina. E o tamanho do arquivo é comparável a uma imagem 1x com qualidade boa. Qual o truque?

Numa tela normal, não retina, a imagem 2x de qualidade baixa vai ser exibida com metade do tamanho e, para isso, o navegador vai redimensioná-la. Nesse processo, o fato da imagem ter o dobro de pixels vai ajudar a obter uma imagem muito bem definida, equivalente a uma imagem 1x de qualidade alta.

Já numa tela retina, todos os pixels da imagem 2x serão exibidos, sem redimensionamento. Mas a tela tem uma densidade tão alta que os pixels são muito pequenos a olho nu. O resultado é que nem notamos a qualidade menor do JPEG. E a imagem fica bem mais nítida que uma imagem 1x tendo seus pixels esticados.

Parece difícil acreditar mas isso foi constatado por vários designers e programadores experientes. Faça o teste na sua máquina acessando o link a seguir. Tente identificar qual é a imagem com qualidade baixa e qual é a com qualidade alta. Melhor ainda se conseguir abrir em uma tela comum e em uma tela retina e comparar os resultados da sua observação.



Figura 19.3: URL do demo: <http://sergiolopes.org/m5>

Retina e PNG

Resolvemos as telas retina com recursos vetoriais e JPEGs 2x com baixa qualidade. Mas e os PNGs? Eles não são arquivos com qualidade controlável como o JPEG, então não é possível adotar uma estratégia semelhante. Como fazer?

Primeiro, tente evitar PNGs. Geralmente eles são usados para gráficos, desenhos, ícones e logos. E tudo isso conseguimos fazer com SVG e icon fonts, que já suportam retina automaticamente.

Mas se você realmente for usar um PNG, vai precisar escolher se vai criar duas versões (uma retina e uma não) ou se vai usar uma versão única 2x. Para usar arquivos diferentes, vimos já os códigos HTML e CSS que você pode usar para servir cada imagem na situação certa.

Eu pessoalmente tenho preferido a estratégia de servir uma imagem única em 2x já. Procuro usar o máximo de SVG quanto posso. E, quando não dá, crio a imagem retina apenas. Tento otimizá-la ao máximo, até convertendo em PNG8, que é menor. Mas evito criar múltiplas versões da mesma imagem. Mas claro que cada caso é um caso e talvez, no seu cenário, seja melhor duplicar as imagens.

Soluções server-side

Muito do trabalho de servir imagens diferentes por causa de formatos, tamanhos de tela ou densidade de pixels, pode ser aliviado com soluções RESS (tópico 18) e técnicas no servidor.

Ao invés de exportar a imagem em vários tamanhos diferentes, você pode ter apenas a imagem grandona em alta definição e deixar o servidor redimensionar para você com base nas características do aparelho. Não vou mostrar código disso exatamente pois varia muito entre as linguagens de programação, mas não deve ser difícil.

Você escolhe uma solução RESS para comunicar ao servidor o tamanho da tela e a densidade de pixels e aí redimensiona a imagem dinamicamente com base nessas informações. Uma solução assim facilita bastante o trabalho do designer que não vai precisar criar cada arquivo na mão, mas tem as desvantagens do RESS que discutimos no tópico 18.

Algumas empresas oferecem serviços de redimensionamento de imagens na nuvem para você não precisar implementar essa lógica no seu servidor. A Sencha tem o **Io Src** (<http://www.sencha.com/products/io>) e ainda há o **ReSrc** (<http://www.resrc.it/>). Você tem apenas que incluir a URL deles em sua página e as imagens são processadas no servidor deles automaticamente.

Uma ferramenta server-side open source bem interessante é a **Thumbor** (<https://github.com/globocom/thumbor>), desenvolvida pelo pessoal da Globo.com. Você pode rodar seu próprio servidor que lida com essas imagens, como o Sencha Io Src. O grande diferencial é que há até uma opção para, ao gerar uma imagem para telas menores, dar destaque para a parte mais importante da foto, recortando-a em vez de só diminuir o tamanho. Eles têm um algoritmo bem esperto de detecção de imagens que calcula o ponto principal da imagem automaticamente.

Conclusão

Imagens responsivas são um tema cabeludo hoje na Web. Ao mesmo tempo que é simples colocar porcentagens e torná-las fluídas, temos o problema de servir imagens diferentes em contextos diferentes. A maneira mais fácil e garantida de fazer isso hoje é com CSS e media queries. Mas pra isso, você precisa usar `background-image`.

Para muitos cenários, você não precisa de uma imagem PNG ou JPG mais. Prefira usar recursos vetoriais como texto, icon fonts e SVG. E lembre dos novos efeitos CSS3 que ajudam a economizar um monte de imagens.

Ao pensar em telas retina, veja também se é realmente necessário disponibilizar todas as imagens também em versão 2x. O logo principal da empresa você vai querer servir sempre com a melhor resolução. Mas muitas vezes você não precisa disso e a imagem 1x é *boa o suficiente*. Em especial quando falamos de fotografias. É preciso testar e tomar decisões conscientes para que o esforço dê mais resultados.

CAPÍTULO 20

Design adaptativo e carregamento condicional

Usuários no celular ou no tablet querem ver o mesmo conteúdo do Desktop. Já discutimos isso nos tópicos de design responsivo (5) e mobile-first (6)

Mas possibilitar o acesso ao mesmo conteúdo não quer dizer disponibilizá-lo da mesma maneira. Há diferenças fundamentais de design e interação do usuário com um celular pequeno e touch e um Desktop grande com mouse. E com tablets, televisões, notebooks etc. Cada contexto de uso pode exigir uma apresentação particular, mas o conteúdo sempre deve estar todo lá.

Quando a tela é grande, por exemplo, é comum disponibilizar um menu grande e completo, com várias opções que se abrem ao passar o mouse. Mas um menu nesse formato no celular é inviável de tão grande e de tantas opções. Talvez você precise de um menu mais focado, menor, com menos opções. Mas, cuidado, isso não significa retirar opções do usuário. Você pode optar por menu com subnível, ou um segundo menu com as opções que foram removidas da parte principal, ou linkar

para as opções secundárias no meio da página mesmo em outra área. De qualquer forma, todas as opções devem estar disponíveis para todo mundo.

O que variamos é a maneira como essas informações são apresentadas. Falamos que **o design foi adaptado** para melhorar a experiência do usuário, mas jamais vamos remover informações ou opções.

Mais exemplos de adaptação

Já passamos por vários cenários de adaptação aqui no livro. Falamos agora de ajustar o menu de acordo com a tela — possivelmente, com um menu diferente e mais focado no mobile.

Quando falamos de mapas uns tópicos atrás, vimos o cenário de mostrar apenas um link para o Google Maps nos celulares e tablets, mas colocar um widget completo no Desktop. Esse caso é até um tipo de adaptação mais agressiva, já que trocamos uma seção inteira da página (o mapa embutido) por um link (que abre a app nativa de mapas). O importante é pensar qual é o conteúdo que está sendo exibido para o usuário e não a forma. E, nesse caso, o pilar do conteúdo é a funcionalidade de ver um endereço no mapa. Se será mostrado via JavaScript num widget (Desktop) ou na app nativa com um link simples, não interessa, é diferença de apresentação. O importante é o usuário sempre conseguir ver o tal endereço no mapa, da forma que for.

Outro caso a se discutir é de *quanto* conteúdo colocar em cada página. Existe uma tendência nas páginas Desktop apresentarem bastante conteúdo que você vai acessando dando scroll. Depois que a comunidade de usabilidade quebrou o mito de que páginas longas são ruins, o mais comum é ver páginas cada vez mais longas com cada vez mais informações.

Mas, claro, informações organizadas e priorizadas. A página longa é eficaz porém deve apresentar o conteúdo com uma hierarquia de importância que faça sentido. Assim, o conteúdo principal deve estar aparente logo de cara (*above the fold*) sem fazer scroll, e o scroll vai abrindo informações secundárias (*below the fold*).

No celular o princípio é o mesmo. Ainda mais pelas telas serem pequenas, o scroll é uma ferramenta essencial para ver mais conteúdo e os usuários estão acostumados a isso. Não há problema em fazer páginas longas no mobile, desde que o conteúdo seja bem pensado e priorizado.

O problema está em pegar aquele conteúdo que já é considerado longo para uma página Desktop e colocá-lo em uma única página mobile. A área é menor no celular

e, por mais que apertemos o design, a página acaba ficando 3x mais longa que a Desktop que já era grande.

O contrário também é verdade: se você pratica *mobile-first*, desenha uma página bacana para celular com todo conteúdo organizado, scroll sendo usado para ver mais coisas, e um tamanho final na página razoável, sem ser muito longa. Mas aí você vai evoluir para o design Desktop e aquela página longa no celular acaba ficando curta demais na versão Desktop, desperdiçando o espaço adicional.

Já vi muita gente criticando designs de páginas responsivas que parecem vazios demais na versão Desktop. Isso, em geral, é reflexo de uma estratégia que priorizou o mobile e acabou usando a mesma hierarquia de conteúdos no Desktop, não aproveitando a área maior. Mas não precisa ser assim.

Conteúdo em várias páginas

Passei por um cenário parecido quando participei do design da página de cursos da Caelum. Cada curso da Caelum tem uma página que mostra as informações do curso. É uma página longa já no Desktop e o desafio era desenvolver uma interface bacana também para os usuários mobile.

O primeiro passo foi identificar quais conteúdos a página mostrava, e chegamos em algo assim:

- Nome do curso e chamada principal;
- Descrição do curso e objetivos;
- Informações de preço, carga horária, pré-requisitos;
- Depoimentos de vários alunos que fizeram o curso;
- Calendário com datas das próximas turmas daquele curso;
- Formulário de contato completo para obter mais informações;
- Ementa detalhada do curso em tópicos;
- E um widget do Facebook pra dar like.

É muita informação. A página Desktop é bastante longa, com quase 5000 pixels de comprimento, exigindo vários scrolls pra se ver tudo. Imagine jogar isso numa tela de celular com um terço da largura: dariam 15000 pixels de comprimento! Inviável.

O próximo passo foi identificar o que era o conteúdo principal e o que era conteúdo complementar. Dois elementos foram considerados importantes mas secundários: formulário de contato na própria página e a ementa detalhada do curso (que é bem grande).

A estratégia foi então separar esses conteúdos em outras páginas e tirá-las da principal. Foi criada uma para o formulário de contato e outra para a ementa detalhada do curso. Na principal, essas seções podiam ser substituídas por links simples levando a esses conteúdos adicionais. No fim, a página do curso ficou bem mais simples e enxuta, essencial para mobile.

A estratégia de quebrar o conteúdo em várias subpáginas é bem interessante para usuários mobile. Páginas mais focadas são mais leves e mais fáceis de ler numa tela pequena. E, ao mesmo tempo, nenhum conteúdo é removido e tudo é acessível ao clicar em mais links.

Mas, para a versão Desktop, não seria necessário quebrar em subpáginas. Pior, se extrairmos o conteúdo para subpáginas, a página principal vai ficar bem menor e pode não aproveitar todo o potencial de área do Desktop. A estratégia pode ser mostrar dois designs distintos: no mobile, páginas menores quebradas em subpáginas; no Desktop, páginas longas com mais conteúdo, sem necessidade de subpáginas.

Carregamento condicional

Uma forma de implementar isso é com Ajax seguindo um padrão que as pessoas chamam de **carregamento condicional** (*conditional loading*). A ideia é simples: o HTML original da página é escrito *mobile-first* e, caso o usuário esteja no Desktop, carregamos os conteúdos secundários.

Ou seja, a página é escrita com o conteúdo principal e apenas com links para os conteúdos secundários. Essa é a experiência padrão nos sites mobile. Aí se detectamos que o usuário abriu num navegador grande no Desktop, carregamos via Ajax o conteúdo secundário que é inserido nela dinamicamente. O efeito final é uma página simples no mobile e uma mais densa no Desktop de maneira transparente.

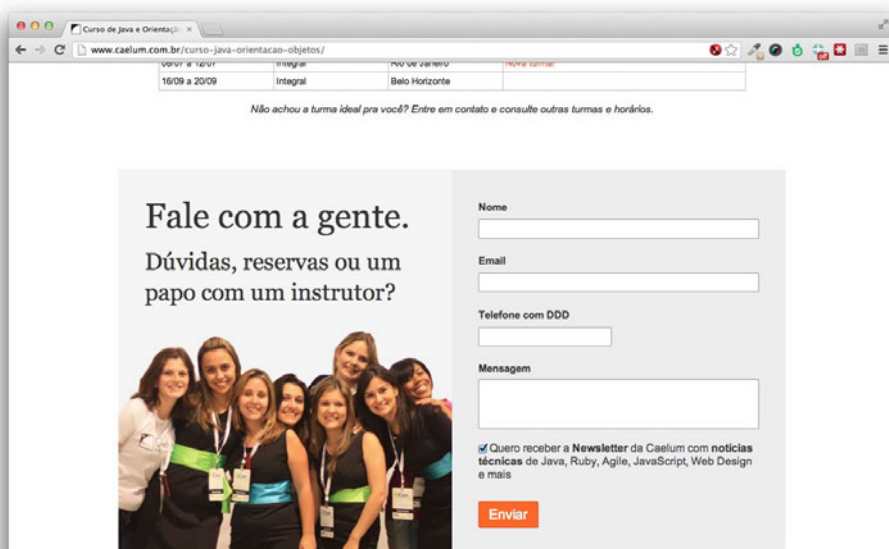


Figura 20.1: A página de cursos da Caelum no Desktop é bem grande. Aqui, destaque pro trecho que mostra um formulário de contato completo.



Figura 20.2: A página de cursos da Caelum num iPhone, com destaque pra parte de contato simplificada com uma chamada e um botão apenas.

Implementando o carregamento condicional

O único truque importante é que o código JavaScript que vai disparar as requisições Ajax precisa identificar se estamos no Desktop ou não, e só então disparar o request. Há várias formas diferentes, como simplesmente pegar o valor do tamanho do viewport (como vimos no tópico 10) e fazer um `if`.

Ou usar media queries para isso, já que provavelmente é o que estamos usando já para as outras adaptações de layout. A vantagem é poder usar qualquer media query existente de forma fácil, e não só de largura de tela. Só precisamos de uma maneira de executar a media query em JavaScript para, então, disparar o Ajax. Existe uma API oficial pra isso, com o método `matchMedia` que funciona na maioria dos navegadores (e ainda tem um polyfill bem simples para os demais: <https://github.com/paulirish/matchMedia.js/>).

O código é bem simples: verificamos a media query e aí disparamos o Ajax (usei jQuery para o exemplo):

```
if (matchMedia('(min-width: 800px)')) {  
    $('#ementa').load('/ementa-curso.html');  
}
```

O método `matchMedia` devolve `true` se a media query em questão bater. Se quiser fazer algo dinâmico, podemos adicionar essa verificação nos eventos de `resize` e `orientationchange` do navegador e fazer um controle mais fino da interface.

No Ajax, o código apenas chama a página secundária e insere o HTML dela direto no elemento `#ementa`, bem simples. Um detalhe só é que você vai precisar expor o HTML do conteúdo secundário em alguma URL (chamei de `ementa-curso.html` no exemplo). É possível usar o HTML da própria página secundária na ementa no Ajax, mas aí você vai precisar se preocupar em extrair só o pedaço com o conteúdo realmente, já que a página completa provavelmente vai vir com header, menu, rodapé etc. Aliás, com jQuery é bem fácil fazer isso. Se o conteúdo estiver, por exemplo, num div interno com id `#conteudo`, podemos pedir pro método `load` carregar só aquele pedaço e desconsiderar o resto da página:

```
$('#ementa').load('/ementa-curso.html #conteudo');
```

Essas soluções são puramente *client-side*, mas dá pra usar RESS (tópico 18) pra otimizar um pouco mais. Se no servidor você já tiver condições de saber qual é o tipo de navegador, é uma boa ideia já mandar o HTML final incluído e economizar um request Ajax. É bem simples implementar uma lógica dessas no servidor e pode ser uma otimização interessante.

E se carregar errado?

Como discutimos no tópico de RESS (18), sempre que trabalhamos com detecção do navegador existe uma margem para que dê errado. Mesmo na solução puramente *client-side* com JavaScript, várias coisas podem dar errado: o JavaScript pode estar desabilitado; o código pode dar uma exception e não executar; o request Ajax pode falhar; a detecção do tipo de navegador pode errar etc.

Mais importante que tentar blindar o código contra erros é encarar o carregamento condicional como *progressive enhancement* e programar defensivamente. Isso quer dizer que a experiência do usuário deve ser boa em todos os cenários.

O fato de usarmos um link simples para uma página secundária já nos protege do cenário onde o JavaScript não executa por algum motivo. O usuário Desktop vai poder clicar no link e ver o conteúdo adicional, assim como o usuário mobile. Um detalhe importante é que essa página secundária não pode ter layout só mobile mas deve ser responsiva e estar preparada para receber usuários Desktop também, mesmo que só para o caso do Ajax dar problema.

O cenário inverso seria carregar o conteúdo a mais do Desktop mesmo na tela mobile por causa de algum erro na detecção. Apesar de a página acabar ficando grande demais, pelo menos não deixe quebrado. Faça o design responsivo funcionar nesse cenário também, mesmo que seja mais raro.

Outro caso é o de um usuário Desktop abrir o site, receber a versão completa e depois redimensionar a janela pra ficar pequena. Você pode usar JavaScript para “descarregar” o conteúdo adicional e mostrar o design mais mobile; ou deixar o conteúdo adicional lá mesmo, uma vez que já foi carregado, e só adaptar responsivamente seu layout.

Perceba, o princípio de se aplicar bem *progressive enhancement* é se precaver em todos os cenários possíveis e prover uma experiência usável mesmo quando as coisas não saem exatamente como prevemos.

Buscas e links

Uma questão importante é como lidar com mecanismos de busca (SEO). Como o HTML é da versão mais simples e com links para páginas secundárias, é isso que o Google vai indexar. Do ponto de vista da indexação não há problemas, já que todo conteúdo será disponibilizado para a busca de qualquer forma. Mas você pode também optar por uma solução RESS, se preferir, e indicar o HTML com conteúdo completo pro Google indexar uma página só. É possível até distinguir os robôs do Google normal e mobile, e dar HTMLs diferentes pra cada um. Só cuidado para, de uma forma ou de outra, todo o conteúdo ser sempre acessível para o Google, senão seu site pode ser penalizado.

A questão maior é o que acontece quando um usuário faz uma busca e clica num link dos resultados. E se o Google indexou as páginas secundárias pensadas para mobile e o usuário busca e clica no Desktop?

Esse mesmo cenário acontece se o usuário usa mais de um dispositivo para navegar. Por exemplo: começa a ver o site no celular, manda um link de uma página secundária e abre depois no Desktop.

É importante, portanto, como falamos antes, que ambas as experiências sejam

sempre compatíveis tanto com mobile quanto com Desktop. É o mesmo caso de programação defensiva de antes. Tanto a página simples, quanto as secundárias, quanto a versão completa devem ser responsivas e adaptáveis para os dois.

Performance

Uma preocupação importante é com relação à performance das soluções de carregamento condicional. Se for pra linha do RESS, não há problemas, mas a versão em JavaScript (que é mais comum) tem algumas implicações.

Do ponto de vista do mobile, o carregamento condicional é uma excelente otimização de performance. Isso porque o usuário não vai receber a página completa e mais pesada, o que é muito bom pensando em redes 3G lentas e aparelhos mais limitados. Mas há um certo sacrifício da versão Desktop.

Pelo menos à primeira vista, salta aos olhos o fato de a versão Desktop exigir mais requests e Ajax, o que pode deixar a página mais carregada. O tempo de relógio para carregamento total da página vai ser maior com certeza, mas isso não necessariamente é ruim para a experiência do usuário. Acabamos usando carregamento condicional para conteúdos secundários que, em boa parte dos casos, está lá embaixo da página e nem é visível inicialmente para o usuário.

Isso quer dizer que se aquele pedaço lá debaixo demorar um pouco mais pra carregar isso pode não ser um problema tão grave. O usuário não vai ver aquilo logo de cara (não recomendo carregamento condicional para conteúdo acima do *fold*). Então a experiência do usuário, que conta mais que o número de segundos no cronômetro, não deve ser prejudicada.

Mais que isso até. Com uma implementação mais esperta, o carregamento condicional pode inclusive melhorar a performance percebida pelo usuário, apesar dos requests adicionais. Isso porque, se a página inicial é mais simples, o navegador renderiza aquilo primeiro e já mostra para o usuário; o conteúdo adicional que nem é visível de início, só vai ser carregado um pouco depois. Ou seja, na prática, a parte inicial da página que é mais importante acaba até carregando mais rapidamente.

Indo mais longe, podemos segurar as requisições dos conteúdos secundários até serem realmente necessários. Se estamos falando de um conteúdo lá embaixo na página, você pode optar por segurar seu download até o usuário fazer scroll. É uma espécie de **lazy load**, que muita gente já usa para imagens, por exemplo. Fazendo esse carregamento atrasado, a performance final sentida pelo usuário é muito boa.

Parte IV

Gestos e entrada de dados



CAPÍTULO 21

Gestos na Web

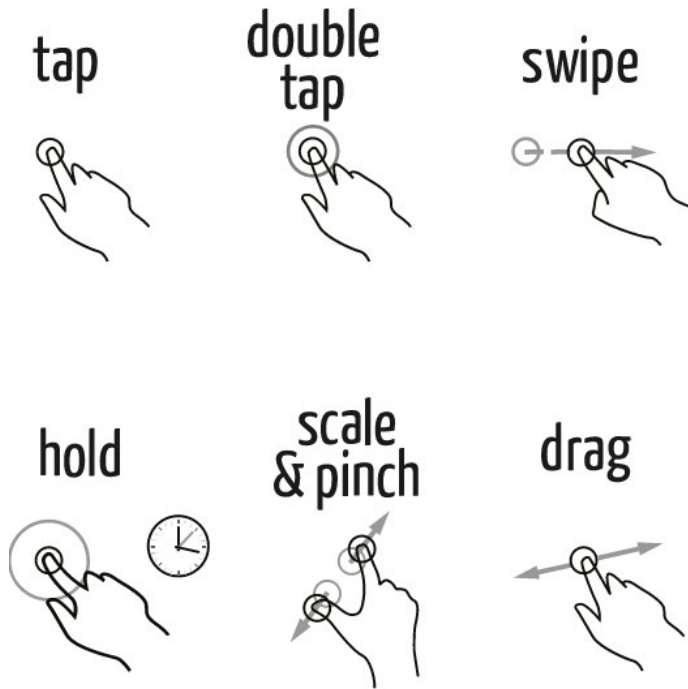
Quais gestos usar

As telas *multitouch* mais avançadas chegam a detectar até 10 pontos na tela ao mesmo tempo. E cada dedo pode ser usado para tocar ou arrastar, criando, na prática, **dezenas de possíveis gestos**. Se você é um usuário avançado do iPad, por exemplo, está acostumado com o gesto de arrastar 4 dedos pra trocar de App, ou encolher 5 dedos pra fechar uma App. Além claro, dos eventos clássicos de toque, duplo toque, zoom etc.

Mas, na prática, o que nós, desenvolvedores Web, podemos usar de gestos?

Os gestos mais comuns na Web são os mais simples, executados com um dedo só ou, no máximo, dois. Muitos aparelhos mais simples não têm tela multitouch e só suportam 1 dedo — vários Androids *low-end* são assim. Além disso, a detecção dos gestos via JavaScript é algo relativamente complicado, o que dificulta a implementação de gestos mais complexos.

Os gestos mais comuns para uso na Web são:



- **Tap** (batida rápida): É o clique do touch. Um toquezinho de leve pra abrir um link, acionar um botão, etc.
- **Double Tap** (duas batidas rápidas, como um duplo clique): Há vários usos particulares em sites e Apps específicos. De modo geral, também é usado para zoom.
- **Hold** ou **Long Tap** (tap demorado): Tocar na tela e segurar. Costuma acionar um menu de contexto daquela ação, como um clique da direita no mouse.
- **Swipe** (arrastar pro lado): Muito usado para navegação de telas, de fotos, em carrosséis, etc. O mais comum é arrastar para a direita e esquerda, embora seja possível fazer swipes verticais também.
- **Scale & Pinch** (movimento de pinça, abrindo ou fechando): É o clássico movimento de zoom, usado para aumentar ou diminuir fotos, páginas Web etc.
- **Drag** (arrastar elemento): Arrastar um elemento na tela de um ponto a outro.

É preciso lembrar, porém, que nossas páginas Web não são Apps nativas, mas vivem dentro de uma, o navegador. E o navegador em si já reserva uma série de gestos para uso próprio. O *swipe vertical* já é usado para scroll; o *pinch* já é usado para zoom; o *hold* exibe um menu contextual que permite abrir links em novas abas ou visualizar uma imagem isoladamente.

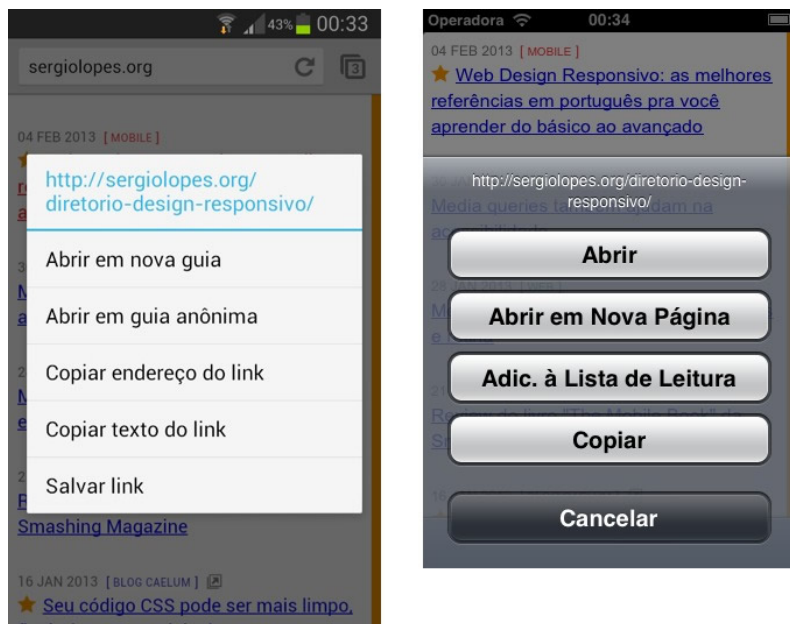


Figura 21.1: Resultado de um hold em cima de um link no Chrome do Android e no Safari do iOS

Você consegue, se quiser, sobrescrever esses gestos do navegador via JavaScript. Mas isso raramente é uma boa ideia. O usuário está acostumado aos gestos padrões da plataforma dele e do navegador dele; seria péssimo comprometer essa familiaridade. Algumas exceções são aceitáveis em cenários bem particulares, como jogos ou Apps como Google Maps (que usa gestos de *swipe*, *drag* e *pinch* pra navegar no mapa).

Na prática, em sites mobile, você encontra muito uso de **tap** e **swipe horizontal** apenas.

Descobrimos gestos

Usar gestos só tem um grande problema de usabilidade: como o usuário *sabe qual gesto está disponível*? Alguns são bem óbvios e fazem parte do básico de todo usuário de touch screens — como *tap* e *scroll*. Outros estão virando padrão e cada vez mais gente sabe como usá-los — como um *swipe* lateral para passar itens, muitas vezes denotado pela presença dos pontinhos:



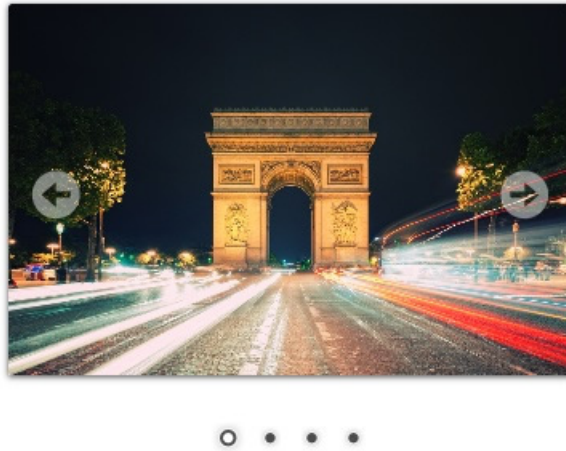
Figura 21.2: Pontinhos comumente usados pra indicar um controle com *swipe*

Mas há zilhões de gestos mais complicados, menos óbvios, que ficam escondidos nas aplicações por aí. Por exemplo, será que todo mundo sabe que um *swipe com 4 dedos* muda de aplicação no iOS? Certamente não, é um gesto escondido. Josh Clark, um famoso autor na área de mobile, escreveu: **gestos são os atalhos de teclado das telas touch**. É uma excelente analogia.

Os gestos são atalhos para você interagir de forma mais simples e natural com seu dispositivo. Mas você precisa saber esses atalhos. Gestos, como os atalhos de teclado, são invisíveis. Não são ações óbvias como acionar um botão. Isso traz duas implicações importantes na hora de criarmos nossos sites e aplicativos.

Primeiro é que você não pode esconder ações importantes por trás de gestos invisíveis. Não faça, por exemplo seu site sem menu nenhum para ter uma interface clean, e com um gesto de três dedos pra acioná-lo. Pensando ainda em Web multi-dispositivos, sabemos que muitos aparelhos nem vão conseguir executar gestos complicados por limitação de hardware. Então faça tudo acessível da forma mais simples possível, com *taps* em botões e menus explícitos. Use gestos como algo a mais, um atalho, ao melhor estilo *progressive enhancement*.

Mesmo no exemplo do *swipe* lateral para passar itens num carrossel: o ideal é não contar apenas com os pontinhos para indicar a presença de mais itens, e não contar com o usuário fazendo *swipe* pra passá-los. Exiba explicitamente controles de *próximo* e *anterior*, bem visíveis, e acessíveis com um simples *tap*:



Às vezes, porém, a experiência central da sua interface é baseada em gestos. Por exemplo, usar o *pinch-to-zoom* para aumentar uma foto ou um mapa. Nesse caso, é importante **ensinar o usuário**. Mostrar pra ele, na primeira vez, quais gestos estão disponíveis e pra que eles servem. Cuidado pra isso não ficar chato. É melhor ir mostrando os gestos aos poucos, conforme ele vai usando a aplicação na prática, no melhor estilo *gamification*.

Essa interface que ensina gestos malucos pro usuário funciona bem em Apps instaladas, mas deve ser evitada na Web. A Web em geral é mais simples e mais voltada para consulta rápida de informações. Não quero ter que aprender uma interface nova, gestos novos, pra cada site que visito. O melhor é a interface familiar e simples.

CAPÍTULO 22

Implementando gestos com JavaScript

Trabalhar com eventos para gestos em JavaScript não é trivial. Existem várias formas de se fazer isso.

Em primeiro lugar, todos os navegadores com suporte a touch suportam os eventos “clássicos” da Web: `click`, `mousedown`, `mousemove`, `onkeydown` etc. Isso, claro, é pra manter compatibilidade com todos os sites criados pra Desktop. Então você consegue implementar um *tap* usando um evento `onclick` clássico.

Mas esses eventos clássicos são limitados quando pensamos nas possibilidades do touch. Com os eventos de mouse, por exemplo, só conseguimos emular um toque único, e não os múltiplos pontos do *multitouch*. Pensando nisso, existem eventos específicos de touch. Os 3 eventos principais são `touchstart`, `touchmove` e `touchend` — outros, mais específicos, incluem `touchcancel`, `touchenter`, `touchleave`.

touchstart e touchend

Imagine implementar um controle básico que muda de estado *ON/OFF* conforme clicado. Visualmente é algo assim:



Para mudar de um estado para o outro, apenas adicionamos uma classe *active* ao elemento via JavaScript, com uma função parecida com essa:

```
function troca(e) {  
    e.currentTarget.classList.toggle('active');  
}
```

Para disparar essa função, podemos usar os eventos. Podemos usar o `click` que funciona em qualquer navegador, inclusive os touch. Ou podemos testar algum evento específico de touch, que pega exatamente o instante em que o dedo toca a tela (`touchstart`) ou o instante exato em que o dedo deixa a tela (`touchend`):

```
document.getElementById('switch').onclick = troca;  
  
// ou  
document.getElementById('switch').ontouchstart = troca;  
  
// ou  
document.getElementById('switch').ontouchend = troca;
```



Figura 22.1: URL do exemplo: <http://sergiolopes.org/m6>

Há vantagens e desvantagens em cada abordagem:

- **click**: Funciona em todos os navegadores, mobile e desktop. Nos dispositivos touch, costuma ser mais lento (em torno de 300ms) pra disparar. Quando tocamos a tela, o navegador espera pra ver que tipo de evento vamos fazer (pode ser um scroll, um double tap, um long tap).
- **touchstart**: Só funciona em navegadores touch e dispara imediatamente após o toque, sendo mais rápido que o click. Se sabemos que aquele elemento nosso não vai aceitar outros eventos, podemos considerar que o touchstart é o disparo de um tap. Mas é difícil garantir isso. Um caso é quando o usuário toca a tela pra fazer scroll; o touchstart vai disparar assim que o dedo passar em cima do elemento, mesmo não sendo essa a ação desejada.
- **touchend**: Parecido com o touchstart, só que dispara no instante que tiramos o dedo da tela.

Se quisermos usar os eventos touch pra pegar a ação de *tap*, provavelmente precisaremos ser mais espertos do que usar apenas o touchstart/touchend sozinhos. Pra detectarmos o *tap* sem afetar as outras ações de *scroll* ou *pinch-to-zoom*, vamos ter que registrar o touchstart e o touchend e tentar verificar se esses dois eventos representam um tap.

Uma forma de fazer isso é calculando a diferença de tempo entre o touchstart e o touchend. Registramos um timestamp em cada evento e vemos se a diferença é pequena (no máximo, 150ms por exemplo). Tocar e tirar o dedo da tela rapidamente é um forte indicativo um *tap*:

```
var start; // vai guardar o tempo de início do toque

document.getElementById('switch').ontouchstart = function(e){
    start = new Date().getTime();
};

document.getElementById('switch').ontouchend = function(e){
    var agora = new Date().getTime();

    // se start foi há menos de 100ms, dispara ação
    if (agora <= startTimestamp + 150) {
        troca(e);
    }
};
```




Figura 22.2: URL do exemplo: <http://sergiolopes.org/m7>

Se quisermos suportar ao mesmo tempo o `onclick` por causa dos desktops, não podemos simplesmente adicionar esse evento. Lembre que o `click` também é disparado nos aparelhos *touch*, então, na prática, acabaria disparando nossa lógica duas vezes. Aliás, quando o usuário dá um toque na tela num navegador touch, são disparados os seguintes eventos, em ordem:

- `touchstart`
- `touchmove`
- `touchend`
- `mouseover`
- `mousemove`
- `mousedown`
- `mouseup`
- `click`

É necessário algum tipo de controle pra evitar que mais de um evento dispare — por exemplo disparar o `click` mesmo após tratarmos o tap com os eventos de touch.

Uma forma possível é colocar `event.preventDefault()` no `touchend`, o que impede a propagação do evento e o disparo do `click`. Mas isso impede também os eventos de touch nativos do navegador e só o nosso evento vai executar. Isso quer dizer que o usuário ficará impedido de dar zoom ou scroll na página se estiver tocando o elemento com `preventDefault`. Daria um trabalho contornar essa situação, então outra forma é simplesmente remover o evento de `click` se tiver touch:

```
if ('ontouchstart' in window) {  
    document.getElementById('switch').onclick = null;  
}
```

touchmove

As interações realmente interessantes numa tela touch vão além do `touchstart` e do `touchend`. A grande estrela é o evento `touchmove`, que é disparado diversas vezes durante o movimento do dedo na tela. Isso, somado à capacidade de obter as coordenadas do toque, e conseguimos implementar eventos complexos como *swipe*,

Trabalhar com `touchmove` é simples:

```
elemento.ontouchmove = function(e) {  
}
```

E, não tínhamos visto ainda, mas todos esses *touch events* que recebemos como argumento, nos dão acesso aos pontos de toque na tela. É um array, já pensando em multitoques. Em cada ponto de toque, podemos obter dados como as coordenadas do toque.

Um código simples para mostrar no console as coordenadas do toque principal durante o movimento do dedo:

```
elemento.ontouchmove = function(e) {  
    var toque = e.changedTouches[0];  
    console.log('X: ' + toque.clientX);  
    console.log('Y: ' + toque.clientY);  
}
```

Agora é usar a imaginação e usar esses valores pra algo útil. Por exemplo, criar um **slider de imagens** no qual você pode navegar arrastando os dedos para os lados (*drag*).



Figura 22.3: Exemplo de slider que vamos implementar com touch.

Para criar o efeito de scroll dentro do slider, criamos um *div* interno (que vou chamar de *container*) com largura bem grande para caber todos os itens que queremos. Posicionamos esse *container* dentro do *slider* logo na esquerda, mostrando só os itens que couberem. No *slider*, um `overflow:hidden` faz os itens sobressalentes ficarem escondidos.

```
<div class="slider">
  <div class="slider-container">
    <!-- vários itens aqui. imagens, links, o que quisermos -->
    
    
    
    <!-- ... -->
  </div>
</div>
```



Figura 22.4: Ilustração de como o container é pequeno e alguns elementos ficam pra fora.

O CSS para posicionar o funcionamento do slider é:

```
.slider {
  /* tamanho visível do slider */
  height: 100px;
  width: 300px

  /* esconde itens que não couberem */
  overflow: hidden;
}
.slider-container {
  /* mesma altura do slider */
  height: 100px;

  /* largura grande, pra caber todos itens */
  width: 1000px;
}
```

O JavaScript que vamos fazer é basicamente deslocar o `.slider-container` pra esquerda e pra direita conforme arrastamos o dedo. Pra isso, vamos usar o evento `touchmove` e *CSS3 Transforms* pra fazer o deslocamento em si. Algo assim:

```
var slider = document.querySelector('.slider');
var container = slider.querySelector('.slider-container');

slider.ontouchmove = function(e){
  container.style.transform = 'translateX(' + deslocamento + 'px)';
}
```

A ideia parece simples, mas a chave é conseguir calcular o quanto queremos deslocar a cada movimento. Podemos obter a coordenada do toque com `e.changedTouches[0].clientX` mas só isso não é suficiente. A funcionalidade do slider é, na verdade, deslocar os itens de acordo com o *quanto arrastamos o dedo*.

Precisamos obter a coordenada inicial do toque via evento `touchstart` e calcular a diferença com a posição do dedo no momento do `touchmove`:

```
var origem = 0;

// calcula o ponto inicial de toque
slider.ontouchstart = function(e){
  origem = e.changedTouches[0].clientX;
}
```

```
slider.ontouchmove = function(e){  
    // calcula o quanto o dedo moveu com relação a origem do 1o toque  
    var deslocamento = e.changedTouches[0].clientX - origem;  
    container.style.transform = 'translateX(' + deslocamento + 'px)';  
}
```

Um outro detalhe que precisa de cuidado é ir acumulando os vários deslocamentos. O usuário pode fazer um *swipe* inicial e depois fazer mais *swipes* pra ir vendo mais itens. Para isso, precisamos acumular o total já deslocado. O cálculo do total em si deve ser feito quando o evento de toque acabar, ou seja, no touchend. E o código final fica assim:

```
var origem = 0;  
var total = 0;  
  
slider.ontouchstart = function(e){  
    origem = e.changedTouches[0].clientX;  
}  
  
slider.ontouchmove = function(e){  
    var deslocamento = e.changedTouches[0].clientX - origem;  
    var posicao = deslocamento + total;  
    container.style.transform = 'translateX(' + posicao + 'px)';  
}  
  
slider.ontouchend = function(e) {  
    total += e.changedTouches[0].clientX - origem;  
}
```

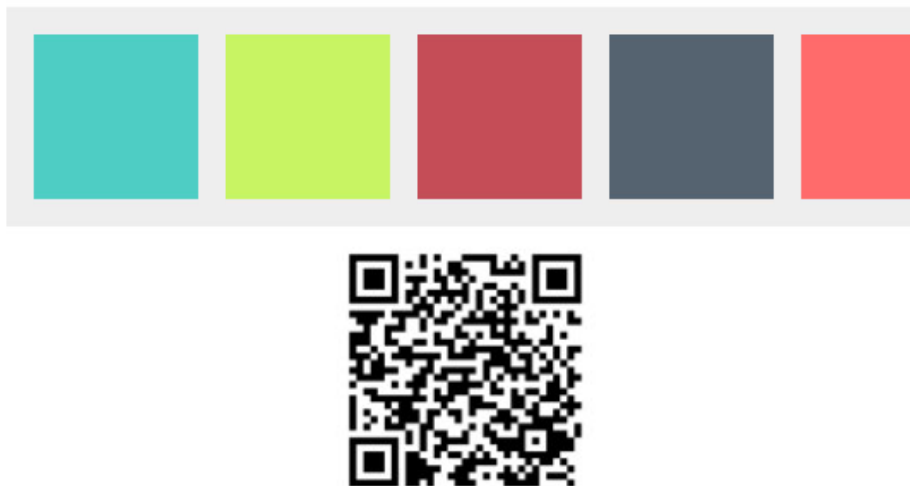


Figura 22.5: URL do exemplo: <http://sergiolopes.org/m8>

Note, porém, que nossa implementação é bem simples e ingênua. Há milhares de possíveis melhorias. Não colocamos limites na coordenada de deslocamento, por exemplo; o ideal seria limitar de acordo com o tamanho do *slider*. Quando o toque acaba, o movimento podia acabar suavemente ao invés da maneira abrupta que está agora (igual o scroll com *momentum* do iOS). E se começarmos a testar nos dispositivos, vamos encontrar muitos bugs e diferenças — o Android, por exemplo, exige um `e.preventDefault()` nos eventos de touch pra tudo funcionar direito.

Além disso, não nos preocupamos em compatibilidade com Desktop e navegadores sem eventos touch. Se queremos um site responsivo compatível com todo tipo de navegador, precisamos tratar também eventos de mouse ou oferecer outra forma de navegação como botões pra passar as páginas do slider.

Libs touch

Trabalhar com eventos touch é JavaScript é difícil. Definitivamente **muito difícil**. Eu quis colocar aqui no livro uma introdução ao *touch events* dado sua importância. Mas fazer coisas usáveis e sem bugs com elas é algo bastante penoso.

Todos os exemplos que mostrei aqui são bastante ingênuos. Na prática, você vai encontrar muitas inconsistências entre navegadores e plataformas diferentes. Não

basta também só tratar os eventos *touch* e deixar os usuários de mouse na mão. E implementar os fallbacks com os eventos tradicionais (*click*, *mousemove* etc) não é trivial, já que eles também são disparados nos navegadores touch.

É preciso pensar ainda em como não interferir nos eventos padrões do navegador — em especial, o zoom e o scroll (e os exemplos desse capítulo têm infinitos conflitos com esses eventos, que não tratamos).

E se você achou a matemática por trás do nosso *swipe* algo fácil de se perder, experimente implementar um evento multitoque. Fazer um simples gesto de rotacionar com 2 dedos envolve mais trigonometria do que a gente gostaria.

Na prática? **Ninguém trata eventos de touch na unha.**

Existem zilhares de **frameworks de eventos touch** que abstraem toda essa complexidade de matemática, bugs dos navegadores, fallbacks pro Desktop e etc. Recomendando *fortemente* que você use alguma dessas opções.

O **Zepto.js**, por exemplo, é uma biblioteca com sintaxe do jQuery que reimplementa boa parte da sua API pensando em dispositivos móveis. Além disso, acrescenta suporte a eventos de touch como *tap*, *longTap* e *swipe*. Usar o Zepto é bem fácil. Um *swipe* fica assim:

```
$('#slider').swipe(function(){
    // um swipe aconteceu, desloca o slider
});
```

O Zepto é um framework completo, pra substituir o jQuery inteiro nos dispositivos móveis. Mas você pode querer apenas um framework para cuidar dos seus eventos touch, sem interferir no resto. O excelente **Hammer.js** faz exatamente isso em apenas 2 KB de JS. Implementar um gesto de *hold* com Hammer fica assim:

```
var hammer = Hammer(document.getElementById('slider'));
hammer.onhold = function(){
}
}
```

Há suporte para todos os gestos que vimos — *tap*, *double tap*, *pinch*, *hold*, *drag*, *swipe*, *rotate*. O melhor é que ele já faz fallbacks pra Desktops automaticamente (por exemplo, disparando um *click* quando você usar o evento de *tap*). Enquanto escrevo esse livro, a versão 2.0 está começando a ser desenvolvida e promete mais facilidades e melhor compatibilidade.

Por fim, mesmo que você não vá fazer gestos complicados na sua página, talvez seja bom usar um framework para cliques. Lembre que os eventos de *click* disparam

em dispositivos touch, mas com um atraso considerável de uns 300ms. Isso porque o navegador precisa esperar um pouco após o toque inicial pra decidir que o evento é um `click` mesmo e não outra coisa (clique duplo, hold, scroll etc).

Esse atraso de 300ms deixa a página sensivelmente mais lenta pro usuário. O segredo é trabalhar então com os eventos de touch para tentar simular os cliques mais rapidamente. Fizemos um teste simples no início do capítulo, mas nossa solução é muito ingênua para uso prático. Se você quiser um framework transparente para disparo de cliques rápidos em touchscreen, recomendo o **FastClick** da *FTLabs*. É só instalar e pronto: você continua usando o evento de `click` mas agora ele é mais rápido e inteligente.

Futuro, Pointer Events API e opiniões

Pra finalizar o tópico, precisamos discutir rapidamente o que nos espera nessa parte de eventos touch no navegador.

Os eventos que vimos aqui foram inventados pela Apple no lançamento do iPhone, e não são um padrão. Vários browsers copiaram o mesmo modelo de eventos (Android, Chrome, Firefox, Blackberry) e o W3C até chegou a extrair uma especificação para padronizar tudo isso. Mas não há consenso geral, em especial no Internet Explorer.

A Microsoft nunca gostou desses eventos de touch da Apple. E com bons argumentos. O grande problema de se criar um evento de *touch* é que ele não está preparado para o futuro — assim como a criação dos eventos de *mouse* no passado nos deixaram nessa posição de hoje. Pra Microsoft, é hora de parar de criar eventos tão específicos e abstrair tudo isso em uma API que funcione com touch, mouse, caneta e qualquer outra coisa do futuro.

Junto com a Mozilla, propuseram uma nova API, chamada de **Pointer Events API**, que abstrai todo tipo de interação baseada em apontar algum elemento — seja tocando a tela, movendo o mouse, etc. Enquanto escrevo esse livro, corre uma briga boa nos bastidores sobre qual API deve prevalecer, e só o Internet Explorer implementa a *Pointer Events API* (e não suporta os *Touch Events* da Apple).

Eu, pessoalmente, prefiro a abordagem proposta pela Microsoft e gostaria de vê-la sendo adotada por todos os navegadores.

Na verdade, minha opinião é de que precisamos de ainda mais um passo nos navegadores: eventos de touch de mais alto nível, que implementem os gestos já prontos (como os frameworks Zepto e Hammer). Assim como temos um evento pronto pra clique duplo (`ondblclick`) e não precisamos ficar calculando isso na

mão, parece-me razoável esperar que um dia existam eventos prontos padronizados para gestos. Pelo menos os mais comuns — *tap*, *hold*, *swipe*, *pinch*. Mas isso parece estar reservado para um futuro *bem* distante, infelizmente.

EVENTOS DE GESTOS NO iOS

O iOS tem suporte a alguns eventos adicionais, `gesturestart`, `gesturemove` e `gestureend`. Eles permitem identificar os gestos de *pinch* e *rotação*. São proprietários e só suportados no iOS.

CAPÍTULO 23

Desafios de UX em interfaces touch

Pra mim, a grande revolução dessa nossa era de smartphones e tablets pós-iPhone foi a **touch screen**. Já existiam telas assim antes, mas a *popularização* delas é o que fez a diferença. Quase tudo tem uma tela touch hoje. E tudo vai ter uma tela touch no futuro.

A grande diferença que veio com o iPhone — e depois todo mundo copiou — talvez tenha sido a precisão e tecnologia inovadoras usadas na tela. Ou talvez tenha sido a construção de uma interface que usa o *touch* pra aproximar o usuário do conteúdo e da ação. Fez com que a interação com o aparelho seja mais natural, quase tão real quanto interagir com um objeto físico.

Mas isso é muito diferente de como interagíamos com nossos computadores e aparelhos antes. Mouses, teclas.

Existem muitas diferenças fundamentais na forma como interagimos com uma interface touch e como interagimos com uma interface via mouse e teclado. É óbvio isso, você percebe quando usa seu computador e quando usa seu smartphone. Em touch, os controles precisam ser grandes pra você tocar com seu dedo gordo e

impreciso (mais sobre isso a seguir).

Com touch, seu dedo, sua mão, seu corpo, são usados para interagir *diretamente* com o conteúdo na tela. É isso que cria a sensação de interação mais natural, pois é bem diferente de usar um equipamento externo como mouse ou teclado. Mas interagir com o dedo cria uma situação curiosa: ao manipular um elemento, a gente **tampa sua visualização**. A mão fica na frente, diferente de quando usamos mouse e sua minúscula seta na tela. Isso tudo tem implicações gigantescas pra interfaces com usuário.

Por exemplo, uma galeria de imagens onde você toca na miniatura pra ver maior: se colocar as miniaturas em cima, não é possível enxergar o que acontece ao tocá-la, pois a mão tampa a tela:



Figura 23.1: Site do Vimeo aberto no iPad. Tem um slider pra escolher vídeo no topo. A mão tampa a tela quando vamos acioná-lo e não vemos o vídeo abrindo no meio da página.

E muitos outros casos. É por isso que eventos de *hover*, além de não funcionarem direito em mobile, não são úteis. Mesmo que funcionassem, o *hover* em geral é feito

para alterar alguma propriedade visual do elemento quando passamos o mouse em cima dele. Passando o dedo, não conseguimos enxergar o que acontece embaixo.

Outra diferença é a forma como seguramos o aparelho touch e interagimos com ele. Um smartphone, pequeno, costuma ser segurado com uma mão só em modo retrato. Nesse cenário, é comum usar o **dedão** para interagir com uma mão só, que é um dedo maior, menos preciso e mais curto. Ou então podemos segurar o aparelho no modo paisagem, onde é comum usar as duas mãos e interagir com os dois dedões. Tudo isso muda radicalmente a facilidade ou não do usuário tocar certos pontos da tela:

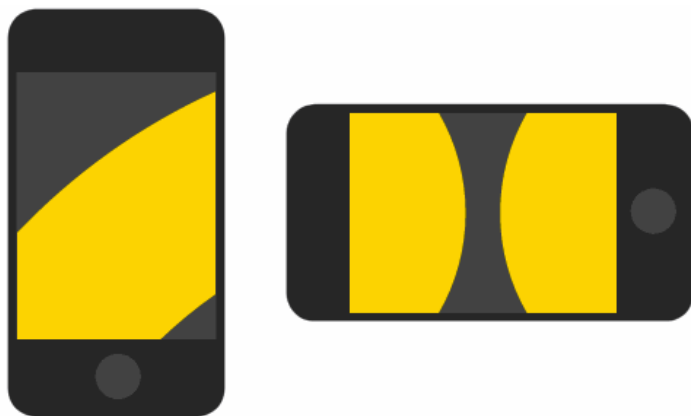


Figura 23.2: Esquema de hot zones para touch em smartphones.

Faça o teste você mesmo. Pegue seu smartphone, segure-o de diversas formas, veja onde seus dedos alcançam com facilidade. É uma forma de interação que traz preocupações que nunca tínhamos pensado no Desktop. O pior lugar pra você colocar um botão importante da sua aplicação é o canto esquerdo superior, difícil de alcançar, se você for destro. Aliás, é por isso que a barra de endereços no IE do Windows Phone fica embaixo da tela:



O mais complicado de tudo isso é que esses padrões de interação mudam de aparelho pra aparelho. Usando um tablet, por exemplo, as áreas de toque mais fáceis de acertar são outras. Usando um notebook com touch ou um tablet apoiado na mesa, tudo muda de novo:



Figura 23.3: Imagem da Microsoft mostrando os pontos de interação com melhor usabilidade para touch nos notebooks híbridos com Windows 8.

Mas é preciso pensar nessas coisas. Existe um motivo pra todos os jogos de computador do mundo usarem as teclas *A*, *S*, *D* e *W* no teclado pra mover o personagem: estão otimizadas pra mão esquerda e próximas o suficiente pra apertarmos com facilidade (imagine se fossem as teclas *G*, *P*, *X* e *M*). No touch é mesma coisa. Precisamos otimizar nossas interfaces para facilitar a interação. Isso envolve os aspectos que citei rapidamente nessa seção e diversos outros que você pode ler em livros especializados

em *interfaces touch*.

O importante é que trabalhar com *touch screens* é repensar a interação e a experiência do usuário. Muito do que fazíamos no Desktop não serve mais.

Detectando touch

Essa diferença brutal na interação entre *touch* e interfaces Desktop tradicionais cria desafios monumentais para quem quer trabalhar com **design responsivo**. Como atender requisitos de interação tão diferentes na mesma página, com mesmo código?

A primeira reação é tentar *adaptar* o design de acordo com o tipo do dispositivo. Já adaptamos a diferentes tamanhos de tela com as media queries clássicas de `width` e `height`, bastaria detectar a presença de uma tela *touch* e adaptar a isso também, certo? Infelizmente, não. Essa detecção não é tão fácil.

A biblioteca universal de detecção de capacidades nos dispositivos, a **Modernizr**, até tem uma forma tentar resolver isso. Ela coloca uma classe no seu CSS com o valor `touch` ou `no-touch`, aí basta escrever o CSS adaptando o design em cima disso. O problema é que essa detecção é extremamente falha.

Não é possível determinar com precisão a presença de uma interface touch. O que o *Modernizr* faz é identificar a presença dos *eventos de touch em JS*, mas isso não é equivalente a ter uma tela touch. Vários aparelhos touch não suportam esses eventos (como os Windows Phone) e vários browsers suportam esses eventos mesmo não tendo uma tela touch (assim como o inverso, de aparelhos touch suportarem eventos de *mouse*, por exemplo). É uma detecção falha e não a recomendo — o próprio pessoal do *Modernizr* tem discussões imensas sobre o assunto.

Existem, porém, outras duas possibilidades de detecção. Uma é detectar no servidor usando algum banco de dados de dispositivos como WURFL — uma abordagem que discuto melhor no tópico 18 sobre RESS. O problema é o banco de dados ficar desatualizado e não identificar adequadamente os dispositivos. A história nos mostrou diversas vezes no passado que soluções baseadas em detecção de *user-agent* são menos confiáveis.

A outra abordagem possível, na verdade, ainda não é muito popular na prática. É um novo valor possível para as media queries que entrou no rascunho da especificação da versão 4 das media queries (<http://dev.w3.org/csswg/mediaqueries4/#pointer>) mas nenhum browser implementa ainda enquanto escrevo esse livro. A sintaxe pode mudar no futuro, mas a nova media query se chama `pointer` e pode receber os valores `fine` e `coarse`, indicando suporte à interface na qual podemos

apontar com precisão (como com mouse ou caneta) ou sem precisão (como o dedo, ou detector de movimentos tipo kinect):

```
@media (pointer: fine) {
    /* ponteiro é preciso, podemos usar botões menores */
}
@media (pointer: coarse) {
    /* ponteiro não é preciso, melhor aumentar os botões! */
}
```

Parece a solução ideal, certo? Além do fato de isso ainda estar num futuro meio distante, não é uma solução à prova de falhas. Os novos notebooks com Windows 8 misturaram tudo isso de novo. São chamados de **híbridos** por trazerem uma *touch screen* e um teclado com trackpad (mouse). E o usuário pode usar ao mesmo tempo o ponteiro do mouse e seu dedo, sem distinção. Nesse caso, a especificação recomenda que o navegador se apresente tendo como base o mais simples — ou seja, será `pointer: coarse`, mesmo tendo mouse além do touch.

Pode parecer um caso particular, mas não é. O Windows é o sistema mais usado no mundo e os novos aparelhos com Windows 8 são híbridos. Isso já seria suficiente, mas eu acho que essa será a tendência para o futuro — aliás, o Chromebook do Google também é um híbrido. Híbridos como esses também serão comuns em outras plataformas, simplesmente porque a *touch* mudou o mundo e se tornará onipresente. Ao mesmo tempo, ainda não estamos prontos pra abandonar o mouse e o teclado de vez, dado que a produtividade e precisão deles é maior que nas interações com toque.

Touch first

Tudo bem, o último parágrafo acima foi um pouco de futurologia da minha parte. É minha opinião sobre os novos aparelhos e, obviamente, posso estar bem errado. Mas pense só no hoje então e esqueça meu chute pro futuro.

Tanto RESS quanto Modernizr são imprecisos pra diferenciar a tela touch de um iPad de um notebook comum. E só tamanho de tela não é suficiente para diferenciar, já que muitos tablets têm resolução idêntica à de notebooks. E, já hoje, existem híbridos no mercado como o Windows 8 que tornam impossível diferenciar uma interface touch de uma “comum”.

Portanto, *essa discussão sobre detectar touch screens é irrelevante*. Ao mesmo tempo, é brutal a diferença de interação das telas *touch* em comparação ao mouse e teclado. O que fazer então? **Touch-first**.

Uma boa interface construída com as restrições do *touch* é completamente usável num Desktop com mouse e teclado. O contrário é que não é verdade. A solução, portanto, é **sempre criar interfaces otimizadas pra touch**, e elas estarão ok nos Desktops.

Isso significa criar botões grandes e espaçados pra facilitar o toque. Preocupar-se com a mão tampando a tela quando fazemos os gestos. Lembrar das áreas mais fáceis de toque de acordo com como seguramos os dispositivos. Pense em tudo isso e sua interface será ótima tanto para *touch screens* quanto para Desktops tradicionais. E se o usuário estiver em um híbrido, poderá usar o mouse ou o dedo ao mesmo tempo, sem dificuldades.

Você vai ouvir falar bastante dessa metodologia *touch-first* na comunidade de desenvolvedores do Windows 8. A Microsoft fala que foi a abordagem usada na criação do sistema, e recomenda essa postura para seus desenvolvedores. Mas o *touch-first* faz todo o sentido na Web, onde temos mais diversidade ainda e é impossível determinar classes de dispositivos (os que tem touch e os que não tem).

Então se vamos implementar tendo touch como base, é bom lidar com algumas diferenças fundamentais.

Touch targets

Uma preocupação importante ao usar touch screens, é com o **tamanho das áreas de interação**. Isso quer dizer que seus botões, links, menus, abas etc, precisam ser grandes o suficiente para o usuário interagir confortavelmente. Isso, claro, porque dedos são maiores e mais imprecisos que um mouse.

Mas qual é o tamanho mínimo recomendado pelos especialistas em usabilidade?

Se você ler na documentação da Apple (<http://bit.ly/apple-touch>), vai ver uma recomendação de, no mínimo, 44px por 44px de tamanho. Muita gente cita esse valor como regra de ouro. Mas é importante contextualizar e não tomá-lo como verdade absoluta.

Quando falamos em tamanho da área de toque, temos que pensar no dedo. Ou seja, não faz sentido discutir isso em *pixels*. O conforto para se usar o dedo deve ser visto em alguma *medida métrica*. Para isso, é bom lembrar do tópico 11 sobre resoluções de tela, DPIs e etc.

Fazendo as contas, quando a Apple fala em 44px num iPhone, ela na verdade está recomendando um tamanho mínimo de **6.8mm**. Outros fabricantes concordam em valores semelhantes. A documentação da Microsoft sobre o Windows Phone recomenda um tamanho mínimo de **7mm**. A Nokia fala em 7mm para uso com o

dedo indicador e 8mm para uso com dedão. A Mozilla fala em algum valor entre **5.9mm** e **9mm**, sendo o 7mm uma boa média (<https://wiki.mozilla.org/Gaia/Design/FlexibleUI>).

A documentação da Microsoft (<http://bit.ly/ms-touch>) é a mais completa nesse quesito, mostrando resultados de um estudo que relaciona tamanho dos *touch targets* e taxa de erros do usuário. Segundo eles, o tamanho médio do dedo indicador é de **11mm**, sendo **8mm** nos bebês e chegando até **19mm** em jogadores de basquete. Com isso, o valor recomendado para nossos *touch targets* seria de **9mm** cuja taxa de erro média é só 0.5% :

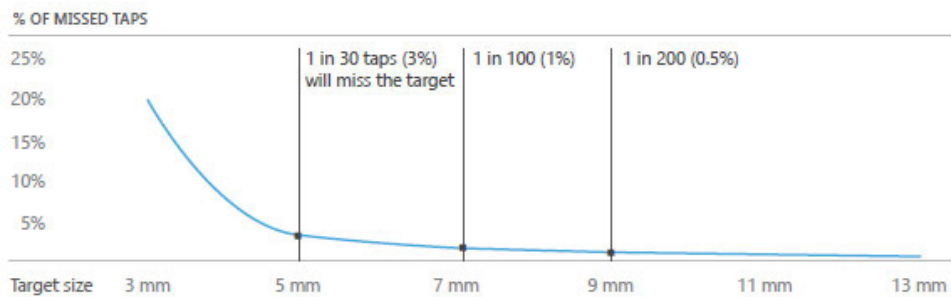


Figura 23.4: Fonte: Microsoft, documentação do Windows 8.

O complicado disso tudo é como, no fim, traduzir para os pixels que vamos usar no CSS. Os 44px recomendados no iPhone, por exemplo, criariam um botão de apenas *5.8mm* no Galaxy 5, um celular Android bastante comum no Brasil. Já num iPad, esses mesmos 44px dariam um botão de *8.4mm*. O problema são as diferentes densidades de pixels nas telas dos aparelhos e é difícil chegar em um valor.

Como não conseguimos detectar a densidade de pixels do aparelho, precisamos de um valor que funcione mais ou menos bem para todo tipo de aparelho. A documentação da Microsoft, ao converter os valores em *mm* para *pixels*, fala num tamanho confortável de **50px**, um valor que eu particularmente gosto. Isso daria *9.6mm* num iPad, *8.3mm* num Galaxy S3, *7.8mm* num iPhone, e *6.7mm* no Galaxy 5. As diferenças são notáveis entre os aparelhos, mas podemos assumir que usuários compram aparelhos que mais se adaptam ao seu uso — é pouco provável você encontrar um jogador de basquete usando o minúsculo Galaxy 5.

Outro ponto a se considerar é o **espaçamento entre os botões**, ou áreas de toque e interação em geral. Não adianta fazer os links grandes, por exemplo, mas

posicioná-los tão próximos que é fácil errar e acionar o errado. Novamente seguindo as recomendações da Microsoft, o ideal seria um espaçamento de **2mm** entre os *touch targets*. Isso pode ser aproximado em **10px**.

Por fim, é bom lembrar que todos esses valores são recomendações. São fortes dicas, baseadas em estudos reais que muitos de nós nem temos como fazer, então é bom ouvir esses conselhos. Mas, claro, você pode abrir exceções conscientes. Porém, é bom ter em mente alguns cuidados adicionais, como usar tamanhos maiores em ações destrutivas, nas quais um erro do usuário seria custoso (apagar uma foto dele, por exemplo). Ou ainda, se for usar botões pequenos, tentar compensar aumentando significativamente o espaçamento entre eles.

Hover

Hover é um caso polêmico. Eles são muito comuns no Desktop mas costumam ser um desastre no touch screen. O que mais existe nos sites Desktop são menus que abrem quando passamos o mouse em cima — no hover. Ou ainda outras interfaces que mostram e escondam conteúdos (um tooltip, por exemplo) baseado no hover.

Tudo isso é bem problemático em telas touch, mas é bom entender o porquê. O mais óbvio é que não existe cursor em telas touch então não existe o evento de se passar algo em cima da página — ou você enfia o dedo lá de vez e conta como um toque, ou você não está interagindo com a tela.

Bem, mais ou menos. Mesmo isso não é verdade absoluta. Precisamos lembrar dos híbridos que têm mouse e touch; eles têm suporte a hover normalmente. Mas mesmo nos smartphones e tablets “puramente” touch também há casos particulares. O Samsung Galaxy Note, por exemplo, tem uma caneta (S-Pen) para o usuário interagir na tela além do touch. E, incrivelmente, essa caneta tem tecnologia para ser detectada sem tocar a tela e, portanto, dispara *hover* normalmente. Há ainda novos smartphones da Sony e da Samsung que usam essa ideia para detectar a proximidade do dedo da tela e disparar o *hover*.

Mais ainda: a pseudo-klasse `:hover` do CSS é suportada em praticamente todos os navegadores móveis!

O iOS, por exemplo, aplica essa pseudo-klasse quando percebe que o site precisa dela pra mostrar conteúdo. O Safari é esperto e vê se você usa `display` ou `visibility` lá dentro e dispara o `:hover` quando o usuário dá o primeiro tap. Eles claramente estão tentando resolver os problemas de acessibilidade dos milhões de sites por aí que usam hover pra esconder e mostrar conteúdo, como menus. Mas essa abordagem tem problemas também. Se for algo clicável, o evento de clique não

dispara no primeiro toque. O primeiro tap é usado pra ligar o hover e é preciso fazer um segundo tap pra disparar o evento mesmo (<http://www.nczonline.net/blog/2012/07/05/ios-has-a-hover-problem/>).

Já o Internet Explorer suporta o `:hover` mas só enquanto o usuário estiver com o dedo na tela em cima do elemento. Assim que ele tirar o dedo o `:hover` sai, parecido com o mouse no Desktop. Mas a maioria dos outros navegadores móveis deixa o `:hover` ativado após o primeiro toque e só o tira quando o usuário toca em outro lugar.

Com relação a JavaScript, há também muitas inconsistências no disparo dos eventos de mouse (como o `mousemove`).

De maneira geral, a recomendação é **não depender de hover para funcionalidades importantes**. Você pode usar o hover pra fazer algum agradinho adicional para o usuário de mouse, mas ele não deve ser obrigatório. Um menu pode ser implementado com clique e oferecer o hover como *progressive enhancement* para os navegadores com suporte. Mas sempre garanta que todas suas páginas funcionem integralmente sem *hover*.

CAPÍTULO 24

Use os novos input types semânticos do HTML 5

Há quem diga que os dispositivos móveis são aparelhos para **consumo de informações**. É verdade, são excelentes dispositivos pra navegar na Web, ler notícias, checar uns emails, ouvir música, ver filmes e assim por diante. Mas a Web de hoje é muito mais sobre **colaboração**: postar nas redes sociais, responder emails, comentar nos sites e blog, editar um artigo na Wikipedia. E queremos fazer tudo isso no celular ou no tablet.

É verdade que digitar um email longo na telinha do celular não é uma tarefa fácil. Mas se engana o desenvolvedor que acha que o usuário não vai tentar fazer isso. Os dispositivos móveis são constantemente usados na **produção de conteúdo**, apesar das suas limitações de usabilidade.

É uma grande contradição. Mas ganha quem conseguir fazer seu site ou App mais usável, apesar da dificuldade natural.

Pensar bem em como seus usuários vão interagir com os campos e formulários

da sua aplicação é essencial. E há alguns truques pra facilitar tudo isso.

Novos input types

Você já criou muitos formulários HTML na vida. Já escreveu vários `<input type="text">`. Durante muito tempo, era isso que tínhamos de opção para criar campos de entrada de texto. No máximo, você fazia um `<textarea>` pra campos maiores.

Claro, o HTML tinha outros tipos, como `password`, `radio` ou `checkbox`, mas todos pra componentes bem específicos. Queria um campo pra digitar texto, seja ele o nome de usuário, seu email ou a data de nascimento? Era `<input type="text">` sem dúvida.

Não mais. O HTML 5 trouxe muitos novos *input types*, como `search`, `email`, `tel` e outros. E o que acontece quando você usa esse algum desses tipos na prática? Veja um screenshot no Chrome:

input text:

input email:

input tel:

input url:



Figura 24.1: URL do exemplo: <http://sergiolopes.org/m9>

Absolutamente nada. Quer dizer, pelo menos, **não visualmente**. Eles continuam sendo campos de texto normais. Mas eles ganharam um novo **significado**. E a parte interessante disso é que está a cargo do navegador interpretar esse novo significado e, eventualmente, prover alguma funcionalidade a mais por causa disso.

Os Desktops e Notebooks têm aqueles teclados chatos, com teclas físicas aparafusadas pra sempre no mesmo lugar. Já as novas fantásticas telas *touch* não têm essa

limitação! Um teclado virtual é desenhado na tela para o usuário interagir. E a parte divertida é que ele não é imutável como os teclados físicos sem graça. E o sistema realmente **muda a cara do teclado** de acordo com o contexto de uso. E como ele sabe o contexto de uso? Interpretando o *significado especial* que demos aos nossos inputs.

Veja a mesma página de antes, agora num iPhone, com foco no campo `<input type="text">` e no campo `<input type="email">`:

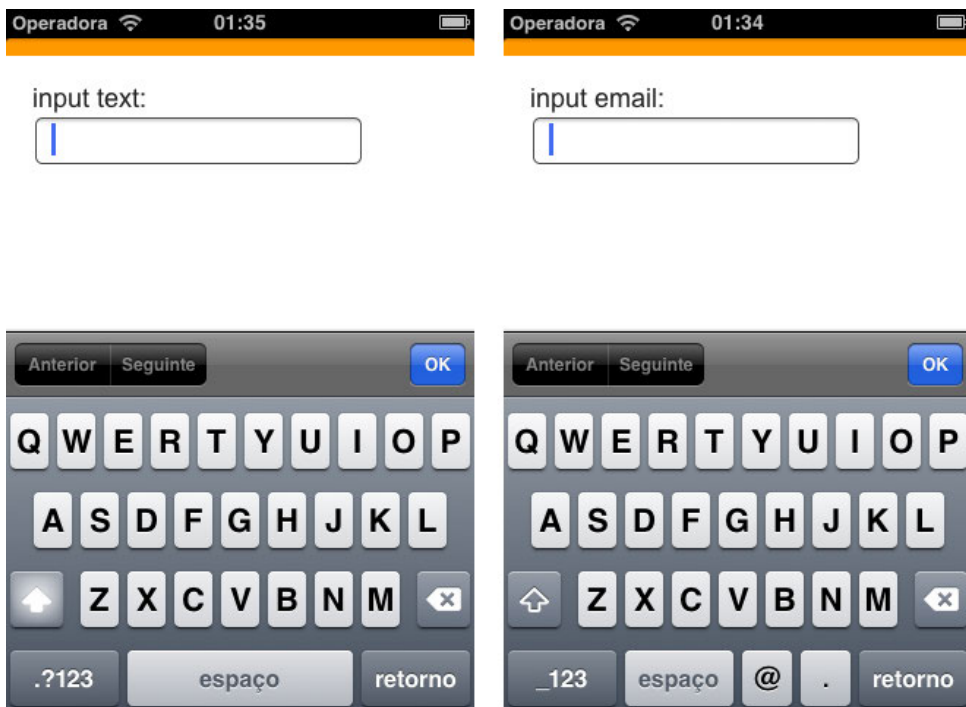


Figura 24.2: Input text e input email no iPhone.

Repare no teclado do iPhone. Diferença sutil mas **essencial para boa usabilidade**. Só colocar a tecla @ lá embaixo já ajuda muito o usuário a digitar seu email. E cada navegador e cada sistema têm a liberdade de interpretar os tipos semânticos do HTML 5 da forma que mais se encaixar.

No Android, por exemplo, o usuário pode instalar teclados customizados. Um deles, que eu gosto muito, o *SwiftX Key* vai além quando encontra um capo do tipo email: ele mostra não só o @, como completa domínios comuns dos usuários terem:

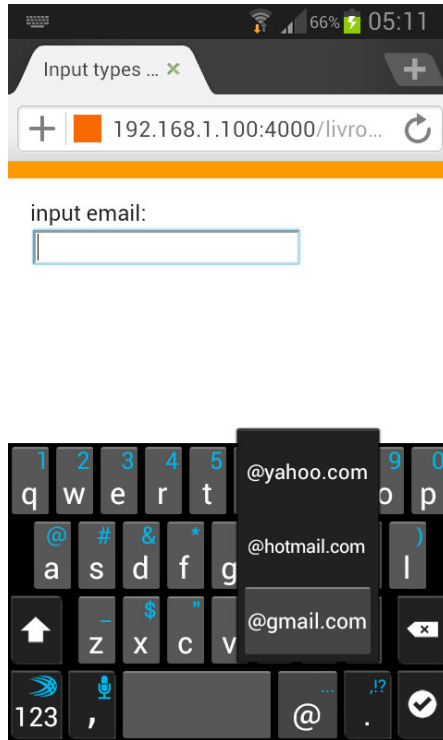


Figura 24.3: Input email no Android com teclado SwiftX Key.

E num Desktop em que não há esse truque do teclado virtual? O campo é renderizado como um campo texto normalmente. Mas nada impede que o navegador melhore a experiência do usuário de outras maneiras no futuro — por exemplo, provendo um autocompletar da agenda de endereços do usuário caso seja um campo `email`.

Isso sem citar que uma **semântica mais definida**, como a desses novos input types, ajuda em outras partes também. o recurso de gerenciar usuários e senhas no navegador pode ficar mais esperto agora que ele sabe exatamente que aquele campo é `email`. Outra melhoria é que o navegador pode ajudar a validar o campo de acordo com regras específicas atreladas àquele tipo. Aliás, nos navegadores que suportam a *Validation API* do HTML 5, é exatamente isso que acontece quando você faz um campo do tipo `email`.

São vários os benefícios de se usar os novos input types, mas vamos nos atrelar às facilidades de usabilidade para usuários em telas touch.

E há muitos outros tipos além do `email`. O tipo `url` ajuda a digitar um endereço na Web e já conta com uma ajudinha no teclado:



Figura 24.4: Teclado no iOS para URL, sem barra de espaços, e com teclas a mais.

Digitar um telefone, então, fica infinitamente mais fácil com um `<input type="tel">`. O teclado se transforma completamente, pra facilitar a digitação de números e separadores:



Figura 24.5: Input tel no iPhone.

Campos para datas e horários

Outro componente importante é o de datas. Durante anos, acostumamo-nos a usar componentes JavaScript para criar um calendário customizado no navegador e evitar que o usuário digitasse a data na mão. Mas isso **não funciona em telas touch**. Quer dizer, o componente até abre e é funcional, mas sua usabilidade o torna *absolutamente imprestável*. Já tentou abrir um calendário de 30 dias todos lado a lado e selecionar um data com o dedão numa tela *touch*? Impossível.

As plataformas móveis resolveram esse problema com calendários nativos mais usáveis. É comum uma interface semelhante a um cilindro que o usuário gira pra escolher a data facilmente:



Figura 24.6: Input date no iPhone.

Imagine recriar essa experiência no nosso antigo calendário em JavaScript. E não é só: no Desktop, você provavelmente vai querer o calendário tradicional já que usar essa versão cilíndrica seria um saco num mouse. Queremos a melhor experiência de usabilidade possível, de acordo com cada plataforma. Como resolver? HTML 5, claro.

Um `<input type="date">` faz tudo isso. Na verdade, faz pouca coisa: só diz pro navegador que queremos entrar com uma data. E o navegador se adapta da melhor forma possível, levando em conta a plataforma, tipo de dispositivo e consistência visual. Veja o mesmo campo date no Firefox Mobile e no Chrome Desktop:

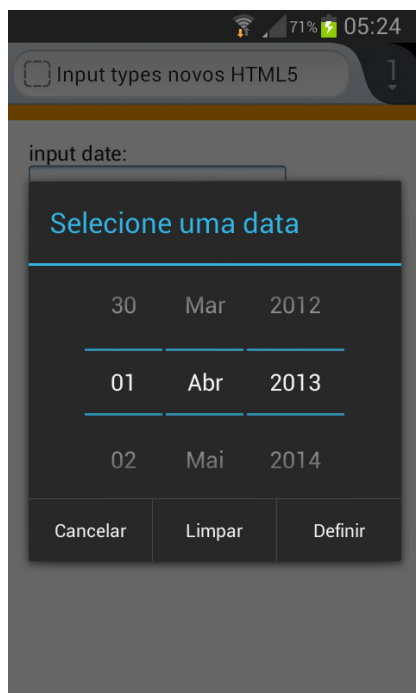


Figura 24.7: Firefox Mobile

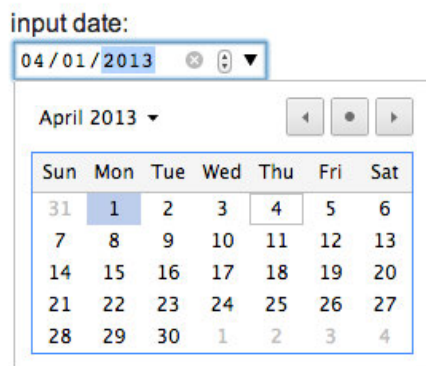


Figura 24.8: Chrome Desktop

Diferenças gritantes, o que é **ótimo**. Talvez você tenha que gastar um tempo explicando para o seu designer, ou convencendo a si mesmo, que a beleza da Web é justo *não controlar os aspectos visuais* desse componente. A usabilidade do componente nativo do navegador do usuário é melhor que a de qualquer calendário customizado em JavaScript, por mais bonito que seja. Isso porque o componente nativo é **contextual e familiar**. Por mais bonito que seu designer ache aquele super calendário rosa choque que ele desenhou, essa não é a experiência mais familiar pro usuário.

E essa customização não é só adaptar o visual à plataforma do usuário. O navegador pode também adaptar radicalmente a interface com base na localização e língua do usuário. Calendários e datas são bem complicados de mostrar em várias línguas; cada *locale* usa um formato diferente, uma notação diferente. O `input date` do navegador cuida disso tudo.

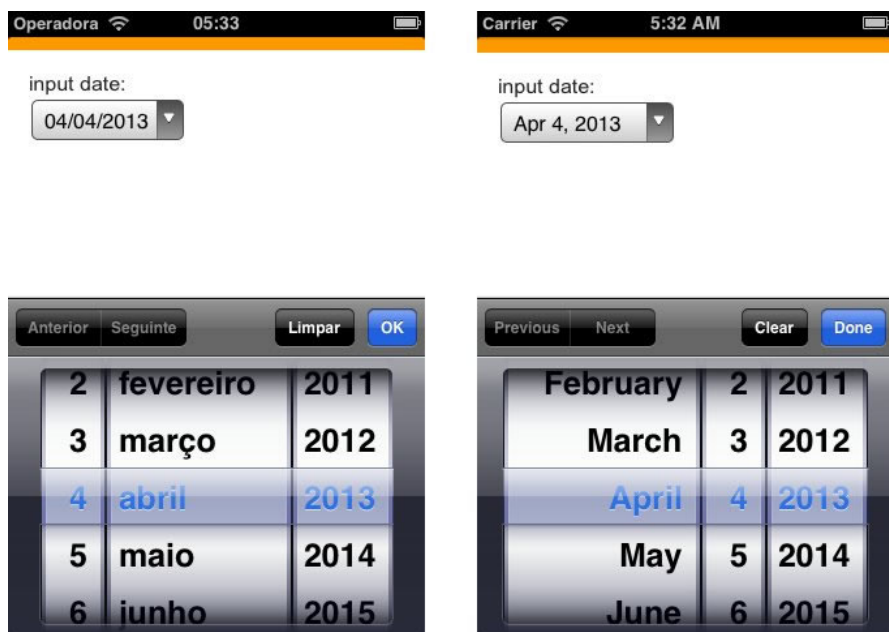


Figura 24.9: iOS configurado em português e em inglês.

Outros inputs relacionados a datas e horários, além do `date`: `month`, `week`, `time`, `datetime` e `datetime-local` — esse último, representando um momento sem estar atrelado a timezones. Você pode testar todos eles no seu aparelho e ver as interfaces na página a seguir:



Figura 24.10: <http://sergiolopes.org/m10>



Figura 24.11: Input datetime no iPhone.

Um último ponto importante é que a especificação garante consistência na representação da data internamente, independentemente do *locale* e da plataforma do usuário. Por baixo dos panos, as datas são representadas sempre em ISO-8601.

Isso quer dizer que um calendário para um dispositivo americano vai *exibir* para o usuário a data como mm/dd/yy, enquanto, em português, será *exibido* como dd/mm/yyyy. Mas o valor que você recebe no servidor ou quando manipular o campo

com JS, será sempre yyyy-mm-dd. Essa normalização facilita muito a vida do desenvolvedor, ao mesmo tempo que garante máxima usabilidade na exibição dos dados para o usuário.

O suporte a todas essas funcionalidades tem crescido rapidamente, mas ainda não é total. Incrivelmente, os navegadores móveis apresentam suporte melhor, talvez pela urgência de se melhorar a usabilidade nos celulares e tablets antes mesmo dos Desktops. Muita coisa já funciona no Mobile Safari do iOS, mas não no Safari do Desktop, por exemplo; mesma coisa com o Chrome.

Para melhorar a questão das incompatibilidades, você talvez queira usar alguma biblioteca JavaScript a mais para mostrar o calendário caso o navegador não suporte os input types novos do HTML 5. Mas lembre-se de fazer isso como fallback; prefira sempre os componentes nativos, se disponíveis.

Números e expressões regulares

Para fazer a entrada de números, como uma quantidade, temos alguns truques. Existe o `<input type="number">` que muda o teclado para a versão numérica nos dispositivos touch:

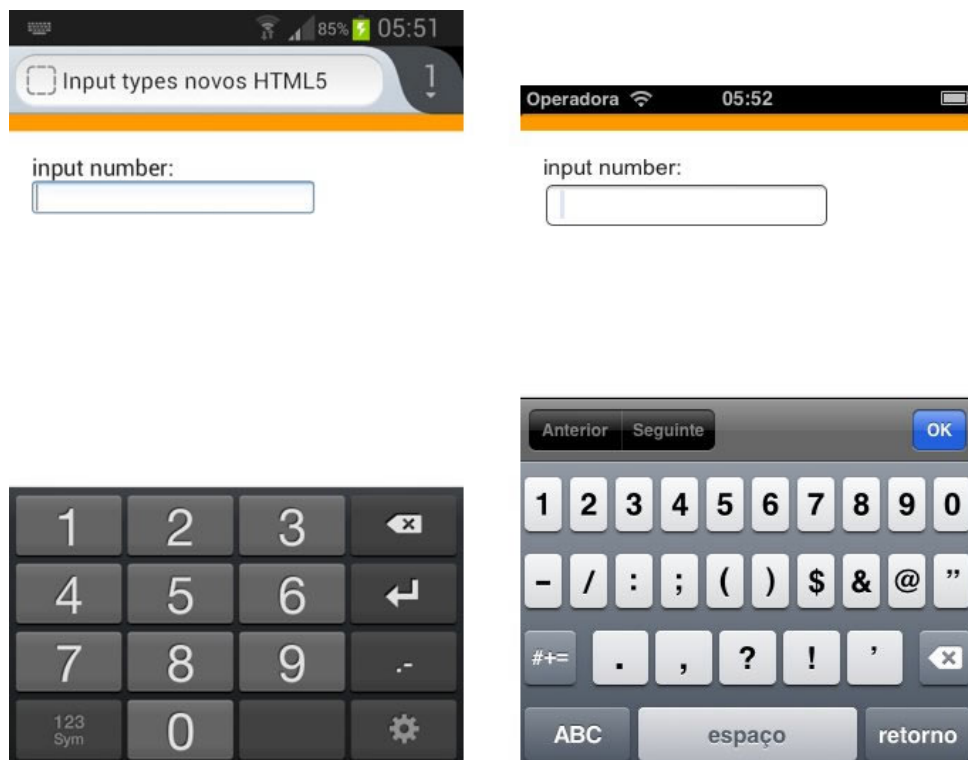


Figura 24.12: Input number no Android (Firefox Mobile no Samsung Galaxy SII) e no iPhone.

No Desktop, é comum aparecer uma interface com setas para aumentar e diminuir o número. É possível também usar os atributos `max` e `min` para controlar o valor, e `steps` indicando quanto deve ser incrementado quando o usuário clicar na seta:

input number:

input pattern [0-9]*:



Figura 24.13: Input number no Chrome Desktop. URL do exemplo: <http://sergiolopes.org/m11>

Se reparar, porém, no teclado do iOS, verá que ele mostra a versão numérica, mas que apresenta símbolos e pontuação também. Existe um truque adicional no iOS que é usar o atributo `pattern="[0-9]*"` e isso vai disparar um teclado focado nos números, parecido com o usado no input de telefone:



Figura 24.14: Input com pattern numérico no iOS.

Esse atributo `pattern` permite que especifiquemos uma expressão regular (usando a sintaxe do JavaScript). Ele foi pensado mais pra parte de validar o formato de um certo campo (é fácil criar um campo para CEP, por exemplo, usando `pattern="[0-9]{5}-[0-9]{3}"`).

Mas, além da validação, o navegador pode usar o `pattern` para ajustar sua interface e o teclado. O caso mais notório é o teclado numérico no iOS que vimos. E, pra falar a verdade, nunca vi outro cenário já implementado na prática. O caso do CEP, por exemplo, vai disparar o teclado normal (nem o numérico). Mas é de se esperar que os navegadores fiquem mais espertos no futuro e consigam até mostrar um teclado baseado apenas na expressão regular.

Para melhor experiência do usuário, portanto, recomendo o uso de um `<input type="number" pattern="[0-9]*">` para números.

Se você quiser trabalhar com valores decimais, é só ajustar o parâmetro `step` para o valor 0.01 - o padrão é 1. Nesse caso, porém, temos que abrir mão do `pattern="[0-9]*"`, senão o usuário no iOS não vai conseguir digitar pontos e vírgulas. O recomendado passa a ser então `<input type="number" step="0.01">`.

E aqui também o dispositivo pode mostrar a interface usando separadores de decimal e milhar baseados na língua do usuário. Mas, internamente, é tudo normalizado para pontos como separador decimal e vírgulas para separadores de milhar.

Mais componentes

Outros novos input types do HTML 5 incluem o `range` e o `color`.

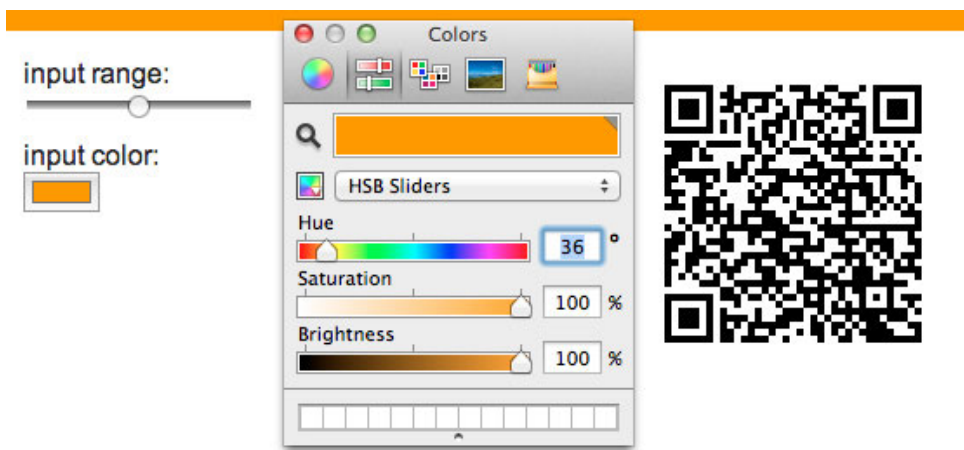


Figura 24.15: Input range e input color no Chrome do Mac. URL do exemplo: <http://sergiolopes.org/m12>

O `range` permite a entrada de um dado numérico com restrição de `max` e `min` e uma interface menos precisa, porém mais simples e fácil de usar. Os dispositivos móveis já começaram a suportar isso, com algumas diferenças. No iOS, o usuário pode arrastar o dedo para selecionar o valor. No Android, ele clica em algum ponto na barra.

Por fim, o `input color` mostra uma interface para seleção de cores, mas seu suporte é bem mais raro ainda nos navegadores.

Atributos iOS

O iOS tem uns truques adicionais que não fazem parte da especificação mas que são úteis e não fazem mal aos navegadores que não suportam. É uma espécie de *progressive enhancement*.

No iOS, você pode controlar se um campo vai ser auto corrigido ou não pelo sistema. Também é possível controlar a maneira como o teclado insere letras maiúsculas no início do campo ou não.

```
<input name="username" autocorrect="none" autocapitalize="on">
```

Por padrão, o iOS corrige e capitaliza todos os campos texto, o que pode não ser interessante em muitos casos. Um campo *username* ou *email*, por exemplo, pode ficar melhor com essas opções desligadas.

CAPÍTULO 25

Revisitando os antigos componentes de formulários

Não é porque o HTML 5 trouxe um monte de novidades nos nossos inputs que não temos que repensar alguns truques dos componentes “antigos”.

Password

O `<input type="password">` é um velho conhecido, mascarando senhas e outros dados sigilosos com asteriscos ou algo do tipo. Digitar senhas em teclados virtuais em touch screen é meio trabalhoso, então os dispositivos tentam melhorar.

Como a chance de você errar digitando naquela telinha é alta, é comum o dispositivo exibir o último caractere da senha por alguns segundos após ser digitado, antes de trocar pelo asterisco. Isso já facilita um pouco.



Mas digitar senhas continua sendo bastante ruim na tela pequena e no teclado impreciso de um smartphone. Por isso, vale a pena repensar esses campos.

Há quem diga — eu inclusive — que devíamos usar simples `input text` em vez de `password`. Mostrar a senha para o usuário sempre, assim ele consegue ver seus erros e corrigi-los. Não preciso explicar para a audiência desse livro, claro, que isso não tem absolutamente **nenhuma implicação em segurança**. Os asteriscos só “protegem” de quem espia você digitando (ou de um vírus que captura sua tela); mas se alguém quiser roubar sua senha, basta olhar você digitando ao invés de olhar pra tela (ou usar um keylogger). Segurança de senhas é criptografia, SSL, só trafegá-la via HTTPS, usar autenticação em 2 fases com tokens etc. E isso independe se a senha é visível na tela ou usa asteriscos.

Embora nós, de uma audiência mais técnica, compreendamos as implicações de segurança, o usuário final pode não ser tão informado. Os asteriscos têm a função de dar uma certa segurança psicológica para o usuário comum — o que, obviamente, já está errado, pois é uma *falsa segurança*. Mas o usuário pode ficar desconfiado se seu site deixar senhas visíveis na tela.

Uma recomendação é incluir um checkbox que permita ao usuário trocar os estados de visível e invisível. Pra implementar, basta um JavaScript simples que troca o `type` entre `text` e `password`. Pro usuário, dá uma segurança a mais. E você pode incrementar isso com frases educativas que explicam que exibir a senha não é inseguro e que ele deve observar o cadeado na barra de endereços.



The image shows a web form with two password input fields. The top field is labeled 'Senha:' and contains five dots, indicating it is hidden. Below it is a checkbox labeled 'invisível?' which is checked. The bottom field is also labeled 'Senha:' and contains the text 'teste', indicating it is visible. Below it is a checkbox labeled 'invisível?' which is unchecked. To the right of the form is a large QR code.

Figura 25.1: Campo senha com checkbox pra deixar visível ou invisível. Na imagem, os dois estados possíveis. URL do exemplo: <http://sergiolopes.org/m13>

Um meio termo interessante é o encontrado no site mobile do Facebook. Para não chocar logo de cara e mostrar a senha direto, o Facebook espera você errar a senha uma vez, e na segunda tentativa te dá a opção de exibir a senha.



Figura 25.2: Interface do site do Facebook mobile quando erro a senha uma vez.

Outra melhoria de usabilidade que você ganha ao exibir as senhas, é economizar um campo no formulário de cadastro. Você não precisa mais do campo de ‘confirmar senha’, já que a função dele é só garantir que sua nova senha é realmente a que você quer. Agora que você vê a senha na tela, não precisa mais da confirmação.

Select nativo

Um componente tradicional que merece um pequeno comentário é o `<select>`. Conhecido como *dropdown* ou *combobox* no mundo Desktop tradicional, ele ganha uma nova cara no mundo mobile.

O iOS mostra o `<select>` com uma interface otimizada pra touch, usando um cilindro para o usuário escolher a opção. O Android mostra uma espécie de popup também otimizada pra touch.



Figura 25.3: Select no iPhone. URL do exemplo: <http://sergiolopes.org/m14>

O `select` deixa de ter a cara de combobox, o que é excelente. De novo, a melhor interface, adaptada por cada navegador para cada situação e plataforma. Isso, claro, se você usar o `<select>` simples e normal do HTML.

Infelizmente, é muito comum encontrar bibliotecas de componente que sobrecrem os combobox com interfaces customizadas. Mais bonitas, com mais funções e mais usáveis no Desktop. Mas podem ser ruins de usabilidade no mobile. Por isso, prefira o `select` nativo sempre que der.

File

Fazer upload de arquivos via HTML não é algo trivial nos aparelhos mobile. Comum no Desktop, a interface de clicar e simplesmente escolher um arquivo não está presente em vários dispositivos. Muitas plataformas, como o iOS, sequer têm um sistema de arquivos para o usuário navegar. No iOS, os arquivos existem dentro de seus programas e você os gerencia por lá — fotos, vídeos, etc. No Android, até existe um sistema de arquivos, mas ele não é exposto muito claramente para o usuário.

Quando você usa um `<input type="file">`, é comum o navegador móvel oferecer opções para o usuário tirar uma foto, gravar um vídeo, navegar na galeria de fotos existente ou algo do tipo:



Figura 25.4: iOS oferece tirar uma foto ou gravar um vídeos nos campos de upload de arquivos.



Figura 25.5: <http://sergiolopes.org/m15>

A interface anterior foi adicionada no iOS 6, e apenas permite envio de fotos e

vídeos (até a versão 5.x do iOS, não existia upload de arquivos de nenhuma forma). No Android, ainda é possível enviar outros arquivos desde que o usuário tenha instalado algum aplicativo que disponibilize isso, como Dropbox ou um navegador de arquivos.

Em geral, portanto, o envio de arquivos em dispositivos móveis gira em torno de **fotos e vídeos**. No HTML 5, é possível até especificar que tipo de arquivo você quer com o parâmetro

```
<input type="file" accept="image/*">
<input type="file" accept="video/*">
<input type="file" accept="audio/*">
```

O iOS suporta as opções de imagem e video, mas não de gravação de áudio. O Android suporta upload há bastante tempo, mas só entende o `accept` a partir da versão 4.

Além disso, há uma extensão ao HTML 5, chamada de *HTML Media Capture* que permite indicar que queremos um arquivo novo, capturado pela câmera ou microfone do usuário naquele momento. É uma espécie de atalho direto pra captura da mídia. Basta adicionar o atributo `capture`:

```
<input type="file" accept="image/*" capture>
```

Enquanto escrevo esse livro, não consegui achar nenhum navegador suportando essa funcionalidade ainda — testei até iOS 6.1, Android 4.2, e navegadores mobile alternativos como Opera, Firefox e Chrome.

O que existe é suporte somente no Android de uma implementação da versão antiga dessa especificação. Na sintaxe velha, a mesma coisa era escrita assim:

```
<input type="file" accept="image/*;capture=camera">
```

O que recomendo é não depender ainda da captura do dispositivo e usar o `accept` simples que vimos antes. Ele é o que trás mais suporte nas plataformas. Se quiser, já deixe o atributo `capture` também, mas pensando no futuro. Eu não usaria a versão antiga que o Android suporta, porque um dia todo mundo vai pra sintaxe nova.

E, com tudo isso, fique atento às versões novas das plataformas e às novidades. Tudo muda bem rápido.

CAPÍTULO 26

Usabilidade de formulários mobile

Além de pensar nos tipos de campos do seu formulário, há algumas outras coisas que você pode fazer pra melhorar bastante a usabilidade deles, pensando mais ainda nos usuários mobile.

Menos campos

Um formulário com menos campos é mais fácil de usar. Óbvio. Mas pense nas implicações disso pra seu site ou sua App. Realmente você precisa daquele monte de campos?

Quando falamos da prática de mostrar a senha ao invés dos asteriscos, comentei que, com isso, você não precisa do campo de ‘confirmar senha’ no formulário de cadastro. Um campo a menos. Será que não conseguimos pensar em outros cenários onde os campos podem ser reduzidos?

Há quem esteja, por exemplo, pensando em unificar os campos de pagamento de cartão de crédito em um só — número do cartão, data de vencimento e código de verificação (<http://zdfs.github.com/toscani/paymentInfo/>).

Todo truque pra diminuir a quantidade de campos é válida. Prefira campos de escolha: checkbox, radio, select, em vez de campos de texto.

Veja se você realmente precisa de todos aqueles dados naquele momento. Talvez você possa repensar seu formulário de compra pra pedir somente informações essenciais no celular, e mandar um email pro usuário preencher campos secundários depois, em casa, com calma.

Seja criativo. Tudo vale pra diminuir o atrito do usuário com sua aplicação. E lembre que input de dados é uma das coisas mais traumáticas.

Geolocation pra endereço

Por várias razões você pode acabar precisando pedir pro usuário o endereço dele, ou algo até mais simples como país ou cidade. E digitar minha cidade parece cada vez mais imbecil quando sabemos que é fácil obter essa informação automaticamente, ainda mais em dispositivos móveis com GPS.

Com HTML 5, é muito fácil obter as coordenadas do GPS do usuário usando a **Geolocation API** com um código JavaScript ridiculamente fácil:

```
navigator.geolocation.getCurrentPosition(function(p){  
  
    var lat = p.coords.latitude;  
    var lon = p.coords.longitude;  
  
    // descobre a cidade a partir das coordenadas  
});
```

Claro, nem todo browser ou dispositivo suporta isso. Mas você pode oferecer como um algo a mais, *progressive enhancement*. Um detalhe é que o usuário precisa **autorizar** a execução dessa API no seu site, por questões de privacidade. A primeira vez que você rodar isso, ele pergunta algo assim:



Figura 26.1: Safari no iOS pedindo permissão pra usar geolocation no site.

Aliás, não é uma boa ideia executar um código de geolocalização automaticamente na página. É muito intrusivo pro usuário. O melhor é deixar ele disparar alguma ação consciente de que isso pedirá a localização dele. Você pode explicar isso na página e usar ícones com os quais ele já está acostumado e já associa com geolocalização

De posse da latitude e longitude do usuário, você vai precisar de algum serviço pra transformar isso num endereço real. Várias APIs de mapas disponibilizam isso, como o Google Maps. Um gratuito e bem fácil de usar é o OpenStreetMaps. Você passa as coordenadas e ele devolve um JSONP bem simples:

```
// jsonp openstreetmap
var script = document.createElement('script');
script.src = "http://nominatim.openstreetmap.org/reverse?format=json"
            + "&json_callback=pegaCidade&lat=" + lat + "&lon=" + lon;
document.head.appendChild(script)

// função que extrai o dado da cidade e seta o input com o valor
```

```
function pegaCidade(data) {  
    $('input[name=cidade]').val(data.address.city);  
}
```

Há muitas outras opções no OpenStreetMaps, assim como na própria Geolocation API. Pesquise e veja o que pode se aplicar a você. Meu intuito aqui é mostrar possibilidades para simplificar seus formulários e melhorar a experiência do usuário.

Aliás, outra opção, caso você não precise do endereço exato (cidade ou país bastam), é analisar o endereço IP do usuário e deduzir daí. Existem serviços com gigantescas tabelas de IP que conseguem te dizer a localização aproximada do usuário. O melhor é que isso não exige nenhuma permissão a mais do usuário, então é um atrito a menos. Eles são menos precisos que GPS, mas podem servir como valor padrão nos campos - o usuário pode editar caso o chute venha errado.

Outros pequenos truques de JS

Assim como a ideia de usar geolocalização pra preencher formulários, você pode ajudar seus usuários com criatividade e vários outros pequenos truques.

Assuma que digitar no celular é ruim e vai dar errado. Prepare sua aplicação para responder a erros. Uma biblioteca JS genial é a **Mailcheck.js** do pessoal da Kicksend (<https://github.com/Kicksend/mailcheck>). Ela oferece sugestões de correção para campos de email com erros comuns - por exemplo, trocar `gnail` por `gmail`. Simples mas genial.

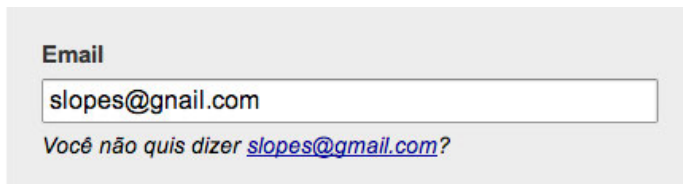


Figura 26.2: Exemplo de uso do Mailcheck.js no formulário de contato do Site da Caelum.

Outro pequeno detalhe que ajuda quando as coisas dão errado: salvar os dados que o usuário vai digitando mesmo antes de ele enviar o formulário. Você pode salvar localmente usando o `localStorage` e recuperar esses dados depois caso algo aconteça com o navegador do usuário: a página pode travar, ou o usuário descuidado pode dar reload sem perceber!

Outra boa pedida são **textareas que crescem automaticamente**. Aquela tela pequena do celular torna difícil a gente colocar um campo de textarea que agrade a todo mundo. O designer quer colocar um pequeno pra não ocupar muito espaço na tela e causar scroll, mas o usuário mais animado pode querer digitar um texto gigante lá. Comum de encontrar em Apps móveis, os textareas que crescem conforme o texto dentro deles aumenta também pode ser implementado em JavaScript com facilidade. Faça isso!

Prefira labels no topo

Existem estudos de usabilidade que mostram que é melhor colocar os labels no topo do campo em vez de do lado. O princípio básico é que o usuário move o olho de cima pra baixo linearmente, ao invés de fazer um zigzag:

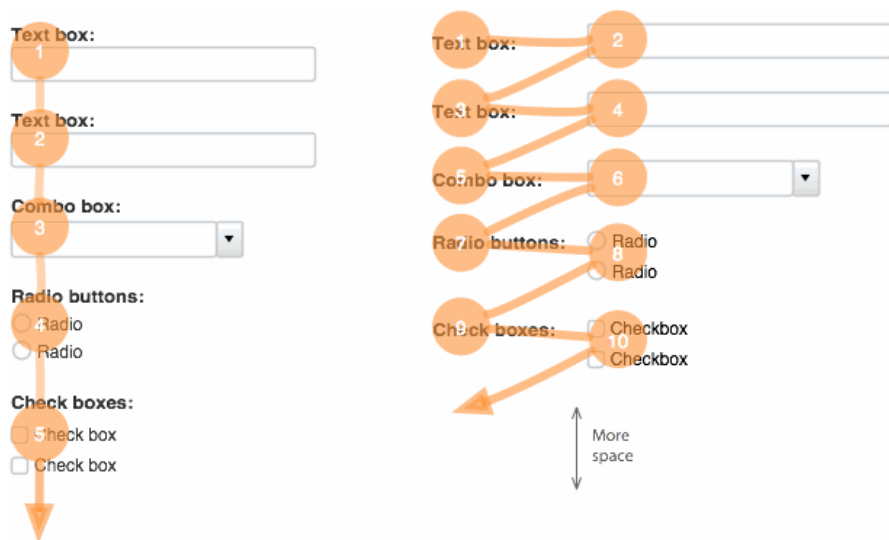


Figura 26.3: Movimento dos olhos ao preencher formulário. (imagem: <http://uxmovement.com>)

Dá pra citar outros motivos que tornam os labels no topo melhores pra Web toda. É mais fácil, por exemplo, usar labels com tamanhos diferentes sem deixar o layout desalinhado. Ou ainda traduzir os labels para outra língua sem correr o risco de quebrar o layout. E mais razões que, por menores que pareçam, fazem sim a diferença na usabilidade.

Mas em mobile? É **essencial colocar os labels no topo**.

Usuários mobile, com aquela telinha minúscula de seus smartphones, estão acostumados a dar **zoom** pra ver melhor a página (e, em algumas situações, o navegador dá zoom sozinho se o campo for pequeno!). Se seus labels ficam na esquerda, eles **ficam escondidos quando o usuário dá zoom no campo**:



Figura 26.4: O exemplo tem um campo com label a esquerda. Dei um zoom na página e repare como o label fica cortado. URL do demo: <http://sergiolopes.org/m16>

O melhor é posicioná-los no topo, onde é mais fácil de ver, mesmo com zoom.

Placeholders não são labels

Falando em labels, muitos designers ficam incomodados com o label no topo ocupar uma linha só pra ele. Isso faz o form crescer verticalmente. Se tiver muitos campos, o scroll será inevitável.

Foi assim que surgiu uma tendência estranha no mundo móvel: a de colocar o label *dentro* do campo. Pior ainda quando isso ficou fácil demais com HTML 5 usando o atributo `placeholder` no input:

```
<input type="text" placeholder="Seu nome">
```

O comportamento de um *placeholder* é mostrar a mensagem dentro do campo até o usuário dar foco ou começar a digitar algo. Aí o texto do usuário toma o lugar do texto do *placeholder*.

Essa abordagem tem muitos problemas. O pior deles, de **usabilidade**, é que não temos mais acesso ao texto do label depois que começamos a preencher o campo. Isso é péssimo. Se o form tem muitos campos (e nem precisam ser tantos assim), o usuário esquece para que servia cada campo. Um usuário desatento (e é o que mais temos em mobile) pode esquecer a função do campo enquanto está no meio dele.

Há outros problemas também, em especial de **acessibilidade**. O placeholder é um valor de exemplo para o campo e não tem a semântica de um label para um leitor de telas. É bem ruim. Até dá pra contornar colocando o `<label>` no HTML e escondendo com CSS, ou implementando a função do label interno via JavaScript ao invés de usar o atributo `placeholder`. Mas ainda temos o problema de usabilidade.

Se você for para uma abordagem de colocar o label interno, pelo menos minimize os problemas. Não use isso em formulários longos — usar naquele form de login com só 2 campos pode ser perdoável, mas não num formulário de 5 campos. E veja se você não consegue adicionar algum lembrete visual do que é aquele campo, como um pequeno ícone na esquerda. Isso já ajuda bastante a usabilidade do seu formulário.

Tamanho adequado para os campos de texto

O Safari no iOS costuma renderizar, por padrão, um campo de texto com fonte 11px. A fonte base do documento (1em) é 16px, mas os inputs por algum motivo são menores.

Só há um problema: se o campo de texto fica pequeno, o Safari dá um zoom nele quando o usuário der foco. Isso pra pessoa poder ver melhor o que está digitando. Mas esse zoom é um problema de usabilidade. É um zoom automático que não é disparado intencionalmente para o usuário.

O ideal é garantir que seus campos tenham fonte de tamanho legível (usabilidade!) desde o início, assim o iOS não precisa dar zoom. É só dar um `font-size` de 16px ou mais e pronto. E, nessa altura do campeonato, nem preciso dizer que a outra solução (desabilitar o zoom no browser) não deve ser feita nunca.

Parte V

Conclusão

É isso. Meu objetivo aqui era mostrar a importância da Web mobile. Quis reforçar o potencial da Web como plataforma única e democrática; o papel fundamental do design responsivo e estratégias como *mobile-first*; a importância de um design adaptado às necessidades do usuário; e um foco na experiência final e na usabilidade.

A gente podia continuar falando de Web e Mobile em mais algumas centenas de páginas e ainda ficaria coisa de fora. Espero que a seleção de temas desse livro tenha atendido ao que você precisava. Se você acha que algum tema importante ficou faltando, não deixe de me contar — quem sabe não saiam outros livros dessas ideias.

Obrigado pela companhia.